MIT/LCS/TR-252

RECOVERY OF THE SWALLOW REPOSITORY

Gail C. Arens

This blank page was inserted to preserve pagination.

Recovery of the Swallow Repository

Gail C. Arens

January 1981

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

Recovery of the Swallow Repository

by

Gail Arens

Submitted to the
Department of Electrical Engineering and Computer Science
on January 26, 1981 in partial fulfillment of the requirements
for the Degree of Master of Science

Abstract

This thesis presents the design of a set of recovery mechanisms for the Swallow repository. Swallow is a distributed data storage system that supports highly reliable long term storage of arbitrary sized data objects with special mechanisms for implementing multi-site atomic actions. The Swallow repository is a data storage server that keeps permanent data in write-once stable storage such as optical disk.

The recovery mechanisms provide on-line recovery for the repository's internal data, as the repository proceeds with its normal operations. In this way, users that wish to access any data that was not affected by the crash can do so while the damaged data is being recovered. Included in the repository's recovery mechanisms are *recovery epochs* and *checkpoint* epochs, which facilitate the detection of damage to the data and minimize the amount of recovery that is necessary. Also included are specialized hash table algorithms that are immune to repository failures. In addition to describing these mechanisms, this thesis discusses how they support the global recovery mechanisms of Swallow and analyzes how they will affect the repository's general performance.

Key Words: distributed data storage system, hash table, recovery, optical disk computer system reliability

Acknowledgments

There are many people who were looking out for my interests throughout my two and a half years at M.I.T. Since I cannot thank them all individually, let this be a general thank you to all those people who are not mentioned below.

I would like to thank my thesis supervisor, Professor Reed, for all of the guidance and assistance he provided throughout the development and preparation of this thesis. Whenever I encountered a problem, no matter how insignificant, he was always willing to help resolve it. His suggestions and criticisms were extremely helpful in solidifying the ideas presented in this thesis.

In addition, I would like to express my gratitude to Frank Vallese, not only for his patience in reading early drafts of this thesis and helping me to express my ideas in a more coherent fashion, but also for being a true friend.

Thanks are also due to all of the members of the Swallow design group, in particular, Professor Svobodova, Dan Theriault and Karen Sollins, who have aided me in my work through numerous discussions of the various ideas related to this thesis. Furthermore, I would like to extend my thanks to Professor Saltzer, who provided the encouragement I needed when I first came to M.I.T.

Finally, I would especially like to thank my parents for the inspiration and guidance they gave me throughout my entire academic career, and would like to thank Jim Chadwick for his constant moral support.

To the memory of my brother, Jesse.

This empty page was substituted for a blank page in the original document.

Table of Contents

Table of Figures

Chapter One

Introduction

As network communications become faster and cheaper it becomes more practical for a single computer, or node, in a distributed computing network to maintain only the resources that it can afford to dedicate, and to obtain all other resources that it may need from other nodes that provide them through the network. In this way, the network provides the benefit of economy of scale through sharing. Long term storage and printing devices are examples of resources that may be shared throughout the network. The nodes that provide the resources are called *servers* while the nodes that share and utilize these resources are called *clients.*

Swallow [16], being developed at M.I.T., is an integrated system of servers that provides reliable, secure and efficient storage for clients throughout a network. The components of Swallow are repositories, authentication servers and brokers. A *repository* is a server that provides very reliable storage for client data in Swallow. It is a processor that is connected to a configuration of storage devices. An *authentication server* acts as intermediary to ensure that all communications within Swallow are secure.[1] A *broker* is a module in the client node that acts as an interpreter for client requests. It mediates interactions between the clients and servers in Swallow. Figure 1-1 shows the general configuration of Swallow in relationship to its clients.

Swallow has several basic features. First, it provides extremely reliable storage. Thus, the probability that any client objects will ever be lost is near zero. Second, Swallow enables the clients to perform any number of accesses (read and write) on an arbitrary set of objects as a single, indivisible (atomic) operation. Third, Swallow protects all objects from unauthorized

[1]The authentication server is not directly relevant to this thesis so it will not be discussed any further. All future references to the *components of Swallow* include only brokers and repositories.

B = broker
R = repository

AS = authentication server

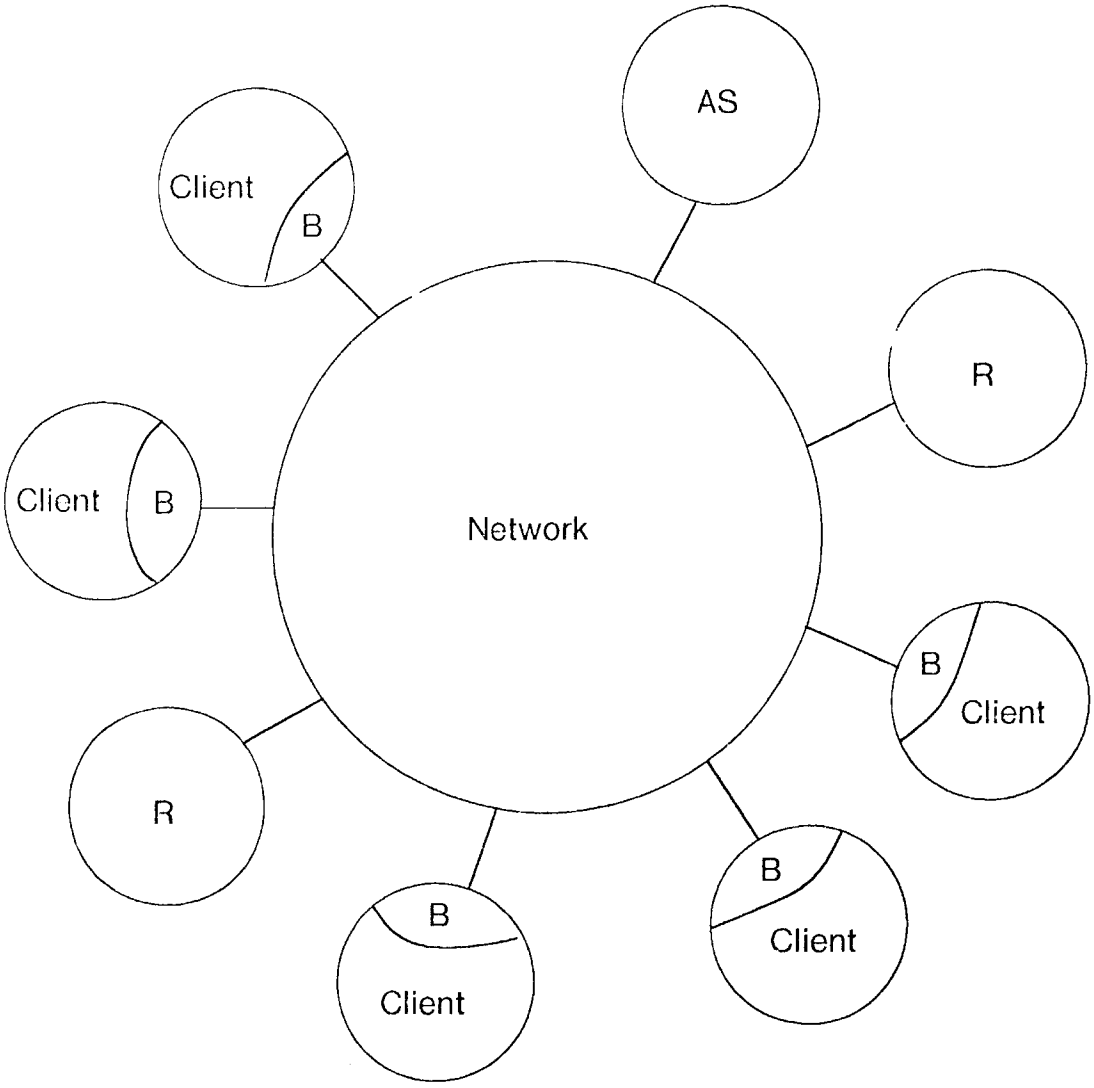


Figure 1-1: Configuration of Swallow

access, using encryption-based mechanisms. Fourth, Swallow, provides a uniform interface for accessing the objects, which may be distributed over a local node and/or several remote repositories. In effect, the clients can specify where they would like each object to be stored, but need not remember the location in order to access the object. Finally, Swallow supports objects of any size, and in particular, very small objects. Thus, Swallow gives the client flexibility in structuring and managing its data, since each object is treated as a separate entity with respect to protection and synchronization as well as with respect to storage and retrieval.

In order to provide these features, Swallow must preserve consistency between all related client data (which may be distributed over several nodes). For example, suppose an appointment scheduling system is a client of Swallow that sets up meetings between people by reserving time slots in their personal calendars. Regardless of where these personal calendars are stored (i.e., in one or more repositories), Swallow must ensure that the calendars are always consistent with one another. In other words, if, as the scheduler is modifying 2 calendars (in order to set up a meeting), the repository in which one (or both) calendar is stored crashes, then either both calendars should reflect the appointment or else neither calendar should reflect the appointment. The state of these 2 calendars, in which only one of them is modified, is internal to Swallow and should never be exposed to the appointment scheduler or any other client that accesses the calendars. Swallow ensures this consistency between related client data by providing a standard set of protocols for all interactions between the brokers and servers, as well as for global recovery. The underlying mechanisms for these protocols and global recovery are based on those developed by Reed [14, 15].

In order for the Swallow protocols and global recovery to be effective, all repositories in Swallow must survive both their own failures and those of other Swallow nodes. This means that all data stored *within* a repository must remain internally consistent, regardless of any errors that may occur due to an internal failure or the failure of another node. For example, within the repository, an object consists of an object header plus the object, itself. In order to update a single object, the repository must modify both the object header and

the object as well as a commit record, which is used to synchronize accesses to the object. Thus, even if the repository crashes in the midst of making these changes, the repository must recover itself to a state in which the object header, object and commit record are consistent with each other, that is, either the state before the update began or the state after the update is completed. In addition, the internal recovery of the repository must support the global recovery mechanisms developed by Reed [14, 15], which restore all related client objects commit records to a consistent state.

This thesis provides the internal mechanisms by which the repository restores its internal state, and integrates these internal mechanisms with the general recovery mechanisms of Swallow in order to show that the recovery of the repository is complete.

1.1 Related Work

WFS [19], Juniper [6] and CFS [1] are other systems that are comparable to Swallow. Each system provides long-term storage in a distributed computing network, but does not have all of the same basic features as Swallow (described on page 10).

WFS was designed to be a more primitive storage system than Swallow. It is a single file server as opposed to a collection of one or more of various types of servers, as in Swallow. Unlike Swallow, WFS does not provide a uniform interface to any data distributed over the local node and the remote file server nor does it restrict access to the data and ensure secure communications. Also, Swallow provides access to objects of any size that do not have to be viewed as standard "files", and provides atomic actions for any arbitrary set of these objects. WFS, on the other hand, provides page level access to files and only ensures atomicity of operations that are executed on a single page (although a system that runs at the client node to provide atomic actions for multiple page and multiple file operations can coexist with WFS [11]).

Juniper is more like Swallow in that it is a distributed data storage system (consists of more than one data storage server) and enables the client to perform atomic actions over multiple data objects at multiple sites, but it still does not have all of the features that

Swallow has. First, Juniper does not provide a uniform interface to data distributed over the local and remote nodes, or to any other types of servers (eg., authentication server). Thus, in order to obtain additional but related services, the client must interface with a different system. Note, though, that plans are in the works to make a system, the Cedar file system, that uses Juniper as a component in a system of structure similar to Swallow. Second, although Juniper provides access to arbitrary sequences of bytes, it does not provide atomic actions for multiple arbitrary sequences of bytes, as does Swallow. In Juniper, the smallest unit that can be treated as a separate entity with respect to an atomic action, is a page. This means that atomic actions can only be performed on multiple pages within a file or throughout several files. In other words, two unrelated data units stored within the same page cannot be accessed in different atomic actions executed at the same time.

The Carnegie-Mellon Central File System project (CFS) is similar to Swallow in that it is a collection of various types of servers that cooperate in order to provide a single, coherent system. Also, CFS makes the location of the data distributed over the local and remote nodes transparent to the clients, as does Swallow. However, the types of servers are not the same in CFS as those in Swallow, and furthermore, the capabilities provided by each system as a whole are quite different. The most fundamental difference between CFS and Swallow lies in the amount of flexibility the client is given for structuring his data. (It is the same fundamental difference that exists between Swallow and both WFS and Juniper). Swallow supports arbitrarily small objects and allows the client to access these objects in whatever fashion suits the particular application. CFS, on the other hand, forces the client to structure and access his objects within the confines a file system. Thus, Swallow provides separate protection for every object whereas CFS only provides protection for files a whole. Furthermore, Swallow provides synchronization for accesses to any arbitrary set of objects (lacking any file structure, within a single file, or within several files) whereas CFS only provides synchronization for access to arbitrary sets of objects within a single file.

The only similarities that exist between the internal recovery for the data storage server in WFS, Juniper, or CFS, and that described in this thesis for the Swallow repository, are that

all of these servers perform their internal operations atomically and maintain any information that is deemed integral to the recovery process in atomic stable storage (except for WFS, which does not support any stable storage). In all other respects, the recovery mechanisms for the Swallow repository differ from those in the storage servers of WFS, Juniper and CFS. Some noted differences are the following. First, the Swallow recovery mechanisms that the repository's internal recovery mechanisms must support are based on mechanisms developed by Reed [14, 15] whereas the other system's global recovery mechanisms are based on other mechanisms [8, 5]. Second, the Swallow repository is the only storage server that uses optical disks as secondary storage. Thus, in Swallow repositories, optimizations in time efficiency are made at the expense of space efficiency, since physical storage is cheap. Finally, the Swallow repository is the only server with append only storage. These, and other differences in the structure and function of the storage servers and the systems as a whole, lead to different requirements for internal recovery of the storage servers, thus, resulting in a unique set of internal recovery mechanisms for the Swallow repository.

1.2 Goals for Repository's Recovery

The repository's internal recovery mechanisms that are presented in this thesis were designed with certain goals in mind. The first and most important goal was to ensure that the recovery mechanisms return the repository to a state in which its data (client objects, commit records, and object headers) are both internally and externally consistent[2] from both the clients as well as the Swallow components' perspectives. This is such an important goal because, as stated before, the general Swallow mechanisms and protocols are based on the assumption that the repositories function properly regardless of failures.

The second goal was to decrease the apparent mean time to repair by minimizing the recovery that has to be done immediately after the repository crashes. Since clients store

[2]Internal consistency refers to the consistency between all related data that is fully contained within the repository. External consistency refers to all related data that is distributed over several repositories.

information in the repositories that they require in order to carry on their regular activities, it is important to minimize the delay that they experience due to a crash. The immediate recovery is minimized by taking advantage of the fact that most crashes affect only a small portion of the repository's data. Thus, the repository restarts as soon as it restores its global state and recovers all client data while receiving and servicing external requests. In this way, the repository allows the clients to access the unaffected data while it is repairing the damaged data.

The final goal was to develop recovery mechanisms that have a minimal effect on the response time for satisfying individual requests, above that which is required to perform the request, since the recovery mechanisms may be in effect while the repository is processing requests. The reponse time for individual requests is affected most significantly by communications and disk transfer delays since the repository is a simple data storage server and most of its work involves transferring the data between the disks and the client nodes. Since the repository's internal recovery mechanisms have very little need for communicating with other nodes, the main way in which they increase the response time is by requiring additional disk accesses. Thus, the recovery mechanisms were designed with the intention of minimizing the additional disk accesses that would affect the response time for satisfying individual requests.

1.3 Outline of Thesis

In Chapter 2 we describe the general mechanisms and protocols that make Swallow a reliable data storage system, and we specify the minimum requirements that individual repositories must satisfy in order to support this reliability. In addition, we summarize the various problems that may affect Swallow's reliability when one of its nodes crashes.

In Chapter 3 we discuss how the repository structures and accesses the data, since it is the data that requires recovery after a crash. In addition, we describe the organization of the various types of storage in which this data is kept.

In Chapter 4 we present the mechanisms that the repository utilizes in order to recover its

data after a crash. For each type of data, we describe how a crash can damage it, and then, how the repository implements its recovery. Furthermore, we justify why some data does not require any recovery at all.

In Chapter 5 we evaluate the recovery mechanisms with respect to performance. We analyze the costs of the recovery mechanisms in terms of their effect on the repository's response time and then compare these effects with the effects that an alternate set of recovery mechanisms (that we could have chosen to use) would have on the response time.

Finally, in Chapter 6 we look back at our original goals and review the strategies that are used to fulfill them. Then we point out several areas where these mechanisms may require improvement and briefly discuss several concepts that can be generalized and used in other systems.

Chapter Two

Overview of Swallow

Swallow is intended to be a very reliable storage system. Basically, it is a set of protocols that allow for proper management of data that may distributed over the local node and several remote repositories. There are various underlying mechanisms that are used in order to implement these protocols. These mechanisms are based on those described by Reed [14, 15]. In order for these mechanisms and protocols to ensure reliability of the system as a whole, the repositories themselves must function properly in the face of failures (both their own, and those of other nodes).

This chapter discusses Swallow as it applies to the repositories. Section 2.1 describes the mechanisms that are used to implement the *atomic action* protocol. Herein, an *atomic action* is defined as well as other terms such as *object history*, *pseudotime* and *possibility*. In Section 2.2, descriptions of the atomic action protocol and several other protocols, on top of which the atomic action protocol is built, are presented. These protocols provide for reliable interactions between repositories and brokers (the two entities that store and manage the data for the Swallow clients). Next, Section 2.3 outlines the minimum requirements that individual repositories must satisfy in order to support the reliability characteristics that Swallow intends to guarantee. (These requirements provided the guidelines for developing the repository's recovery mechanisms). Finally, Section 2.4 lists the general types of problems that can occur when a Swallow node crashes.

2.1 Swallow Mechanisms

In Swallow, the functional unit of client data is called an *object.* Further, the fundamental requests that a client can submit to Swallow (through a broker) to be performed on an object are:

Create Object: writes a new object into storage

Delete Object: eliminates an object from storage

Read Object: returns the current value of an object in storage

Modify Object: assigns a new value to an object and writes it into storage

In addition, a client can submit (through the broker) a series of these requests to be performed as a single *atomic action* [8, 9, 14, 17] by bounding the series with *Begin Atomic Action* and *End Atomic Action* requests.

An atomic action is a set of operations (requests) that must satisfy the following two requirements:

1. *failure atomicity requirement* - the operations of a single atomic action should either be performed to completion or not be performed at all (i.e., aborted if completion is not possible).

2. *concurrency atomicity requirement* - the operations of single atomic action should behave *as if* they are executed serially with respect to the operations of other atomic actions even though atomic actions may be executed concurrently.

To satisfy the failure atomicity requirement, an atomic action is structured so that at some point the atomic action is *committed,* which means that it is irrevocably required to finish. In other words, if there is a failure before the commit point and not all of the component requests have been satisfied then, upon recovery, the system's state must be backed up to the state it had before any of the requests were fulfilled. On the other hand, if the failure occurs after the commit point, then any of the component requests that were not satisfied before the failure occurred must be satisfied upon recovery. To satisfy the concurrency requirements, it is arranged so that the intermediate state of the system during the execution of an atomic action (when only some but not all of the requests have been satisfied) is

protected from any processes performing a different atomic action.

For example, consider the appointment scheduling system described in the previous chapter. The system would be implemented so that the scheduler would request that Swallow read and update several people's calendars as a single atomic action. Then, even if one or more of the repositories (containing the calendars to be modified) crashes, the calendars would either all reflect the scheduled meeting (if the crash occurs after the commit point) or else none of them would reflect the meeting (if the crash occurs before the commit point). Also, if one or more of the calendars does not have the requested time slot open, then the appointment scheduler can explicitly abort the atomic action and none of the calendars would be updated to reflect the meeting. Finally, if several such atomic actions were executed simultaneously, and requested the same time slot in several people's calendars, then one of these atomic actions would appear to execute first and thus, succeed whereas the other would find that the requested slot was filled.

The remainder of this section summarizes the mechanisms developed by Reed [14, 15] that are used in order to implement the atomic actions defined above.

Pseudotimes are numbers that are used to assign a total ordering of events in Swallow. Pseudotimes do not directly correspond to real time. A global clock mechanism supplies a unique, non-overlapping range of pseudotimes, or *pseudotemporal environment*, to every atomic action. Each request that accesses an object is assigned a pseudotime from the pseudotemporal environment of the atomic action.

Objects are implemented in the form of *object histories*. An object history is a sequence of *versions*. Each version is a state that the object has assumed at some point in time. See Figure 2-1. Each version of an object history is valid for a range of pseudotimes. For example, version B in Figure 2-1, is valid from pseudotimes 5 to 10.

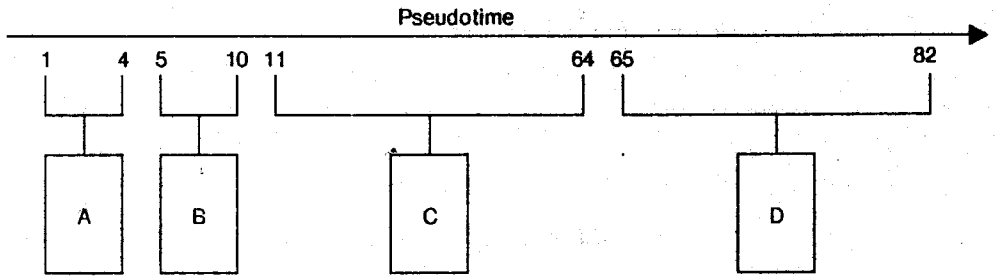


Figure 2-1: Example of an Object History

A *modify* request creates a new version in the object history. The pseudotime of the modify request provides the *start pseudotime*, which is the lower bound for the version's range of validity. If a version already exists in the object history at the pseudotime specified in the modify request, then the modify request is denied. For example, a version could not be created at pseudotime 8 in the object history illustrated in Figure 2-1 since version B exists for that pseudotime.

A *read* request selects the version that has the largest start pseudotime less than the pseudotime specified in the request. Then, the upper bound of the version's validity is extended, if necessary, to include the pseudotime of the read. According to Reed [14, 15], the upper bound of a version is the last pseudotime at which a request read the version. This means that there can be pseudotimes in the middle of an object history for which no versions exist. For example, if a modify request wishes to create a version in the object history shown in Figure 2-1 at pseudotime 90, then version E would be created with a lower pseudotime of validity of 90 and no version would exist for pseudotimes 83 - 89, as shown in Figure 2-2. To simplify matters within Swallow, it has been decided not to leave any holes in an object history [18]. Therefore, when a new version is created at a specific pseudotime, the previously current version's upper pseudotime of validity is extended to the pseudotime at which the new version is being created. Referring back to the previous example, the upper pseudotime of validity for version D would be extended to 89, as shown in Figure 2-3 instead of leaving a hole, as in Figure 2-2.

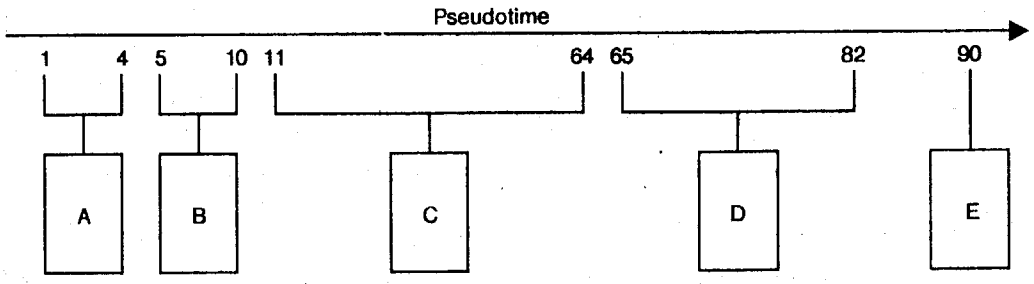


Figure 2-2: Creation of a New Version as Described by Reed

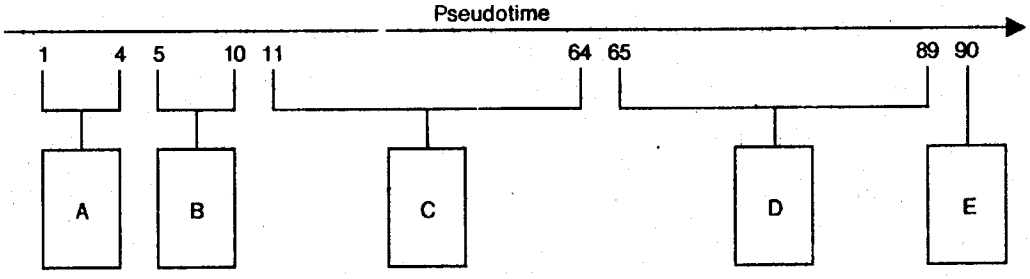


Figure 2-3: Creation of a New Version in Swallow

An atomic action ensures that a specified sequence of read and modify (as well as create and delete) requests for one or more objects are performed as an indivisible unit. If any of the requests are not successfully satisfied, then the atomic action is aborted. Abortions are made possible by making the versions created by an atomic action tentative until the atomic action is explicitly committed. These tentative versions are called *tokens* and are not readable by other atomic actions. In other words, if some request within an atomic action attempts to read a token created by another atomic action, then that request will be delayed until the atomic action that create it either commits or aborts. Upon commiting, the tokens made by an atomic action become versions.

All tokens created by a single atomic action are grouped into a set called a *possibility.* When all of the component requests of an atomic action are satisfied, the atomic action

commits its possibility. This committing converts all of the tokens into actual versions. If, on the other hand, some of the requests are denied, then the atomic action aborts its possibility, which deletes the tokens from the object history.

Possibilities are implemented using *commit records* that record the state of an atomic action. Initially, the state is *unknown*. All tokens in a possibility (or versions, once the possibility is committed) contain a reference (pointer) to the commit record associated with the possibility. Tokens are distinguished from versions by the state of their commit record. When the state of the commit record is changed to *committed* the tokens become versions and can be examined by other atomic actions. If the state of the commit record is changed to *aborted* then the tokens are deleted. Further, commit records must have timeouts associated with them so that if a failure occurs that causes the commit records to neither be committed nor aborted (this could happen, for example, when a client node crashes), then the tokens will not become permanent fixtures in object histories, blocking future real operations on that object. Possibilities enable Swallow to ensure that if an atomic action cannot be completed then the state of the data will appear as if none of the component updates were done.

2.2 Swallow Protocols

In order for Swallow reliably to satisfy the requests submitted by the clients, brokers and repositories must interact in an orderly fashion. The broker must interpret a client request and, in turn, generate requests that can be understood and fulfilled by the repositories. The brokers and repositories communicate their needs to each other by sending and receiving messages, which contain either *requests* or *responses* to some request. Swallow provides standard protocols for sending and receiving these requests and responses under normal circumstances. In addition, these Swallow protocols specify provisional actions that should be taken if the status of communications between two nodes is disrupted by a crash of one of these nodes.

The Swallow Message Protocol (or SMP) described in Section 2.2.1, provides for the

reliable transport of the messages through the network by detecting transmission errors that may occur. The request/response protocol, discussed in Section 2.2.2, provides a guarantee to the requestor that its request has been received and fulfilled. The atomic action protocol, discussed in Section 2.2.3 ensures global consistency of the data distributed over more than one node as well as ensuring that atomic actions behave as if they are executed serially.

2.2.1 Swallow Message Protocol

Every Swallow message is sent through the network in the form of one or more packets. Each packet has a sequence number that indicates which part of the message it contains so that the complete message can be reconstructed at the receiving node. Swallow Message Protocol, SMP, is a very simple protocol that specifies exactly how node A, for example, must send the packets of a message to node B. The protocol is as follows:

1. A sends 1st packet of message
2. B sends back a packet indicating that A can send X number of packets more
3. A sends X number of packets
4. B sends back a packet indicating that A can send Y number of packets more
5. A sends Y number of packets
6. etc.

This continues until the entire message is sent. If either node does not hear from the other one within a reasonable amount of time then it aborts the message and discards any remaining packets. Notice that this protocol is very simple for single packet messages because no connection has to be established. For multiple packet messages, though, it allows the receiving node to exert some flow control so that its buffers don't overflow.

Currently, SMP is built on top of the User Datagram Protocol (UDP) [12]. UDP doesn't resequence the packets of a single message at the receiving node nor does it prevent their duplication. Therefore, SMP is responsible for reordering them and discarding all

duplicates so that the receiving nodes do not have to perform these tasks. SMP does not prevent out of sequence or duplicate *messages*, though, nor does it guarantee delivery of the messages. These problems are taken care of by the atomic action and request/response protocols, respectively.

2.2.2 Request/Response Protocol

Since a requestor can never be certain that its request was received and/or satisfied unless it receives a confirming response [2], there is an associated response for every request sent in Swallow. The response either confirms both the delivery and the fulfillment of the request or rejects the request. If the requestor does not receive a response within a reasonable amount of time then it can retransmit the original request or abort the transmission. The table in Figure 2-4 enumerates the various types of requests and associated responses that can be sent and received by the repository. The next section describes what actions are taken when these requests are received.

2.2.3 Atomic Action Protocol

The atomic action protocol specifies exactly how the brokers and repositories should cooperate in order to carry out atomic actions for Swallow clients. The broker manages the local data, monitors the atomic action as a whole and decides whether to commit or abort the atomic action. On the other hand, the repository stores and manages the object histories and commit records. That is, it reads and writes the actual data and carries out the final phase of the atomic action, in which tokens are converted into versions or are deleted.

The objects updated by an atomic action may be entirely contained within a single repository or distributed throughout an arbitrary number of them. In order to minimize the number of external messages that have to be sent to the repositories, committing or aborting a possibility, each repository that contains tokens whose commit records reside in another repository, maintains a single *commit record representative* for each commit record of an atomic action. A commit record representative contains the state of the atomic action (unknown, committed or aborted), as well as the references to any tokens (created by the

Figure 2-4: Repository Requests and Responses

REQUESTS	RESPONSES	COMMENTS
1. Create-Object	Object-Created	Response contains uid of object (OID)
2.Delete-Object	Object-Deleted or Can't-Delete-Object	Can't-Delete response indicates a synchronization conflict
3. Read-Version	Version-Value	Response contains version valid as of given pseudotime
4. Create-Token	Token-Created or Can't-Create-Token	Can't-Create-Token indicates a synchronization conflict
5. Test-Commit-Record	State-Is: Committed or Aborted	Response contains state of commit record
6. Abort-Commit-Record	State-Is: Committed or Aborted	If commit record already comitted then returns State-Is: Committed
7. Commit-Commit-Record	State-Is: Committed or Aborted	If commit record already aborted then returns State-Is: Aborted
8. Add-Reference	Reference-Added	Request is sent to commit-record-representatives
9. State-Is: Committed or Aborted	Delete-Reference	Request sent to broadcast final state of commit record. Response confirms that final state was encached in commit record representative

atomic action) that reside in the same repository in which the commit record representative is located. Thus, the actual commit record need only maintain references to each repository that contains tokens created by the atomic action rather than to each individual token, as illustrated in Figure 2-5. Further, when a repository has to broadcast the final state of a commit record so that the tokens can be converted into versions or deleted from their object histories, it has to send only one message per repository regardless of how many tokens each repository contains. Then, each repository can act upon all tokens from that atomic action that are referenced by the commit record representative.

Sections 2.2.3.1 through 2.2.3.6 describe the protocol for each type of request that the client may submit.

2.2.3.1 Begin Atomic Action

When a client begins an atomic action, the broker must send a message to some repository, requesting the creation of a commit record. The repository creates it and returns a response which contains the name of the commit record. Once the broker receives this confirmation it can send to any repositories any sequence of create, read, modify or delete object requests, depending upon the client's needs. All of these subsequent requests must include the name of the commit record as well as a pseudotime, so that the repositories can identify the atomic action of which the request is a part and can synchronize all concurrent accesses to the same objects.

2.2.3.2 Create Object

When a client wishes to create an object, the broker sends a create-object-history request to the repository. Upon receiving the request, the repository creates all of the internal structures needed for the object history in storage. Included is a reference to the specified commit record or its local commit record representative.[3] If neither exists in the repository

[3]Both the creation and deletion of objects are also requests that belong to a possibility, that is, if the atomic action creating (deleting) the object fails, then the creation (deletion) is not done.

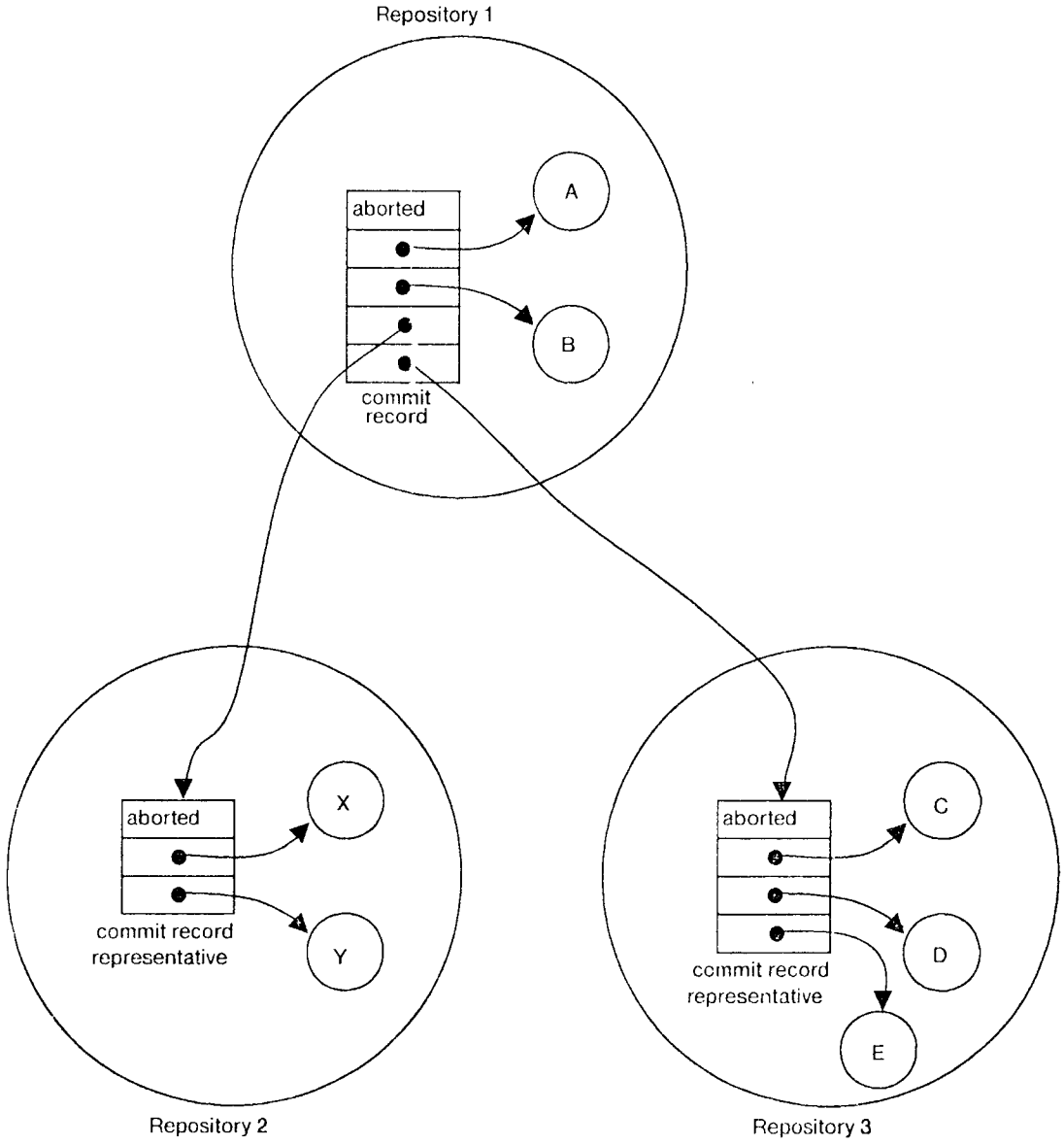


Figure 2-5: Representation of A Distributed Commit Record

at that time, then the repository must create a representative with the correct reference to the version and must send an external request to the remote repository that contains the commit record, asking it to add a reference in the commit record to the commit record representative. Once the local repository receives a response confirming that the reference has been added then it must return a response to the broker, confirming the creation of the object history.

2.2.3.3 Delete Object

When a client wishes to delete an object, the broker sends a delete-object-history request to the repository. When the repository receives the request, it checks whether or not any versions exist for a pseudotime greater than or equal to the one specified in the request. If any exist, then it returns a negative response indicating that the object cannot be deleted. If none exist, then the repository creates the final version of the object history that marks it as being deleted, including a reference to the commit record (or representative) and returns a response to the broker that confirms the object history's deletion.

2.2.3.4 Modify Object

When a client wishes to modify an object, the broker generates a create-token request and sends it to the repository. Upon receiving the request, the repository checks to see if a version already exists at a pseudotime greater than or equal to the one specified in the request. If one exists, then it returns a negative response indicating that the token can't be created at the given pseudotime. If none exists, then it creates the new token, adds a reference to the commit record or representative and returns a response to the broker, confirming the token's creation.

2.2.3.5 Read Object

When a client wishes to read an object, the broker sends a read-version request to the repository. Upon receiving the request, the repository must check whether or not the

version referenced by the pseudotime in the request is a token, an aborted token, or a committed version. If it is a committed version or a token that was created by the same atomic action that sent the read request, then it simply returns that version or token in the confirmation. On the other hand, if the request is for a token that was created by a different atomic action than the one that sent the request, then the repository must check the token's commit record to see whether or not it has been committed. If so, then the repository must commit the token, extend its validity time to the pseudotime specified in the read request and return that version in the response to the broker. Otherwise, if the commit record has been aborted then the repository must abort the token, extend the validity time of the current version to the pseudotime specified in the broker's request, and finally, it must return that version in the response to the broker.

2.2.3.6 End Atomic Action

If all of the component requests of the atomic action are confirmed then the broker finishes the atomic action by sending a commit request to the repository in which the commit record is stored. That repository then commits the commit record and returns a positive response, marking the completion of the atomic action. On the other hand, if the broker received any rejections to its requests then it may abort the atomic action by sending an abort request to the repository, which must then abort the commit record and return a response to the broker, confirming the abortion of the atomic action.

Once the final state of a commit record has been recorded, the repository storing the commit record must broadcast this state to all of the repositories for which the commit record has references. When each repository receives the state of an atomic action it must encache that state in the commit record representative and return a response indicating that the its reference can be deleted from the commit record's list of references. When the commit record has no more references it can be deleted.[4]

---

[4]Note, that this description of the final phase of the atomic action (that is carried out by the repository) has been simplified by ignoring the commit records of nested atomic actions. (See [Reed78])

2.3 Reliability Requirements for Individual Repositories

Now that the global mechanisms and protocols have been described, the two minimum requirements that individual repositories must satisfy in order to ensure reliability of Swallow, as a whole, can be defined as follows, in Sections 2.3.1 and 2.3.2.

2.3.1 Data Integrity

Since the repository stores the clients' objects as well as the commit records that are used to synchronize access to those object, it must protect these objects and commit records against any damage, loss, or inconsistency that may occur when it crashes. In other words, the repository must protect the integrity of all objects and commit records.

In protecting the integrity of the client data, the repository must do more than just ensure that this data isn't lost or damaged. It must also ensure that the objects and commit records are managed properly. This means that a crash should not alter the repository in any way that would cause it to overlook the most current version or token of an object history or create a version at a pseudotime for which a version already exists. It also means that a crash should not cause a repository to release the value of a token outside the atomic action in which the token was created.

2.3.2 Atomicity of Requests

In addition to protecting the data integrity, a repository must satisfy all requests atomically. That is, the multiple internal modifications that must be done as part of a single request, must be done as an indivisible operation. This internal atomicity supports the more general atomicity guaranteed by Swallow to its clients. In the same way that Swallow guarantees not to leave client data in an inconsistent state, a repository must guarantee not to leave its internal data in an inconsistent state.

For example, a version of a large object will span over more than one disk page. If the repository crashes before it writes out all of the pages to the disk and these pages are not written atomically, then the object history of which the incompiete version is a part will be

invalid. Thus, upon restarting, the repository must ensure that the incomplete version is not included in the object history.

As another example, a create-token request involves both recording the new version and adding, to the associated commit record, a reference to the new version. If these two internal tasks are not performed atomically then the Swallow mechanisms for providing clients with the ability to execute a set of requests atomically will not work properly, since the repository will never know whether the token should be converted to a version or deleted from the object history.

2.4 Summary of Problems Caused by Failure of a Swallow Node

We have seen how Swallow ensures reliable storage of the data by providing the client with the ability to execute atomic actions and by insisting that its repositories satisfy several requirements. Before getting into the details of the repository, let us briefly list the general problems that might occur when a Swallow node crashes.

1. *Global (or external) inconsistency of data* - The related client objects stored throughout Swallow may not be current with respect to one another. The atomic action protocol ensures consistency with the support of the repositories, which properly maintain and manage all commit records.

2. *Internal inconsistency of data within the repository* - The objects, commit records and other data supporting these objects and commit records may not be consistent with each other within the repository. The repository's internal recovery mechanisms restore internal consistency of the data, as will be described in this thesis.

3. *Out of sequence packets w/in a message* - Communications delays may cause packets of a message to arrive in a different order than which they were sent. SMP resequences these packets.

4. *Retransmitted packets w/in a message* - A node sending a request may retransmit packets if it thinks that the original packets were lost. SMP discards duplicate packets.

5. *Unconfirmed messages* - A message may not be acknowledged if the receiving node crashes. The combination of all three protocols and the repository's

internal recovery mechanisms ensure recovery of any damage caused by unconfirmed messages. How they ensure this will be clarified in this thesis.

6. *Incomplete messages* - A repository may not receive all of the packets of a message if it or the sending node crashes. An incomplete message does not get confirmed so it is recovered as an unconfirmed message. This problem affects the repository since the data of a large object version is written into stable storage as received, before the complete message is available.

7. *Out of sequence messages* - Due to the distribution of the nodes and real time delays, requests may not be received in the same order that they are sent. The atomic action protocol serializes all requests by using pseudotime instead of arrival order.

8. *Retransmitted messages* - If a node does not receive a confirmation for a request, it may retransmit the request. All requests that can be send to the repository are *repeatable*; that is, the repository will make the requested modifications in response to the same request only once (the repository can recognize retransmitted requests). Upon receiving a retransmitted request, the repository simply confirms it and does not repeat the modifications that are requested. This thesis will demonstrate how the repository properly handles retransmitted requests.

This thesis deals directly with problems 2, 5, 6 and 8. More discussion on the other problems above will found in [14, 15, 16].

Chapter Three

Management of Data within the Repository

The repository's data can be classified as follows: object data, commit record data, pending messages data, and data that describes the repository's global state. In order to understand how the repository recovers this data after a crash, it is first necessary to explain the internal structure and management of these four classes of data as well as the organization of the storage in which the data is maintained.

Sections 3.1 and 3.2 describe the object and commit record data, which consist of sequences of versions plus a header that contains a reference to the current version. Next, Section 3.3 discusses the message data, which consists of sequences of packets. Then, Section 3.4 briefly describes the global state data, which is a record that describes the status of the repository as a whole.

The remaining sections describe the various forms of secondary storage that the repository supports as well as their interaction with primary storage. Section 3.5 gives an overview of the organization of the storage in the repository and then Sections 3.6, 3.7 and 3.8 describe Version Storage, State Storage, and Object Header Storage, respectively.

3.1 Objects

Within the repository, an object is represented by the versions of the object history plus an object header, which contains a reference to the current version and other useful information about the object. Figure 3-1 illustrates the internal structure of an object.

Thus, in order to create a token (assuming that no token already exists) the repository creates a version (as depicted in Figure 3-1) in storage, and then modifies the object header, as follows. The value of the token reference is changed from nil to the newly created

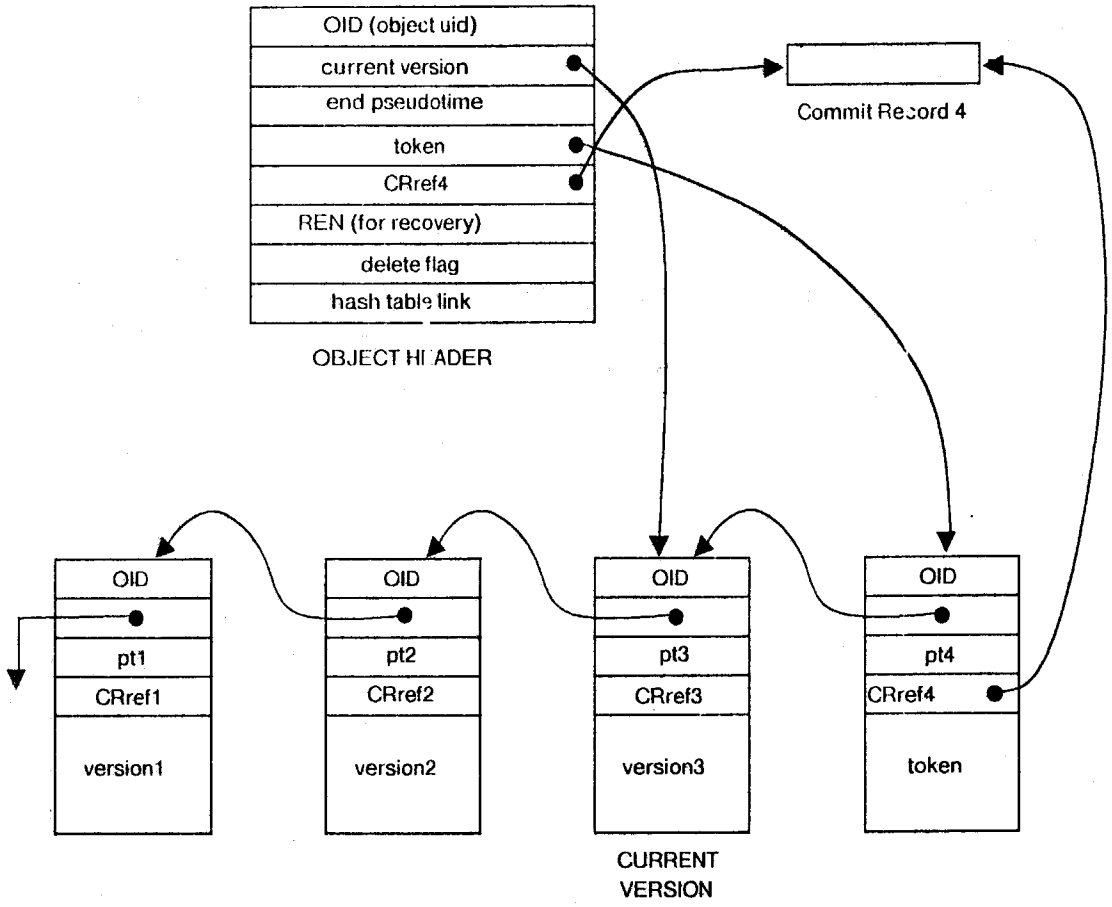


Figure 3-1: Structure of an Object Within the Repository

token's address in storage, the value of the commit record reference is changed to the unique identifier of the token's commit record, and the end pseudotime value is changed to the pseudotime at which the token is created. Subsequently, if the token becomes a version, the repository changes the references within the object header: the value of the current version reference is changed to the token's address in storage, and then the value of the token reference becomes nil. Alternatively, if the token becomes aborted, then the repository deletes it by simply changing the values of the token reference and commit record reference in the object header to nil. Finally, in order to read a version of the object, the repository obtains the location of the current version in storage from the object header.

Since the objects are accessed using the object headers, the repository organizes the object headers in the form of a hash table, called the *object header table.* This object header table will be discussed in more detail in Section 3.8

3.2 Commit Records

Conceptually, a commit record consists of the state of the atomic action that it represents, and a list of references to the tokens created by that atomic action. Within the repository, a commit record's structure is similar to that of an object. A commit record (or commit record representative) is structured as a threaded sequence of versions. Furthermore, the repository maintains a hash table, called the *commit record table*, whose entries contain the state of the commit record and a reference to the current versions of the commit records.

Figure 3-2 depicts a commit record after the atomic action's final state has been decided.

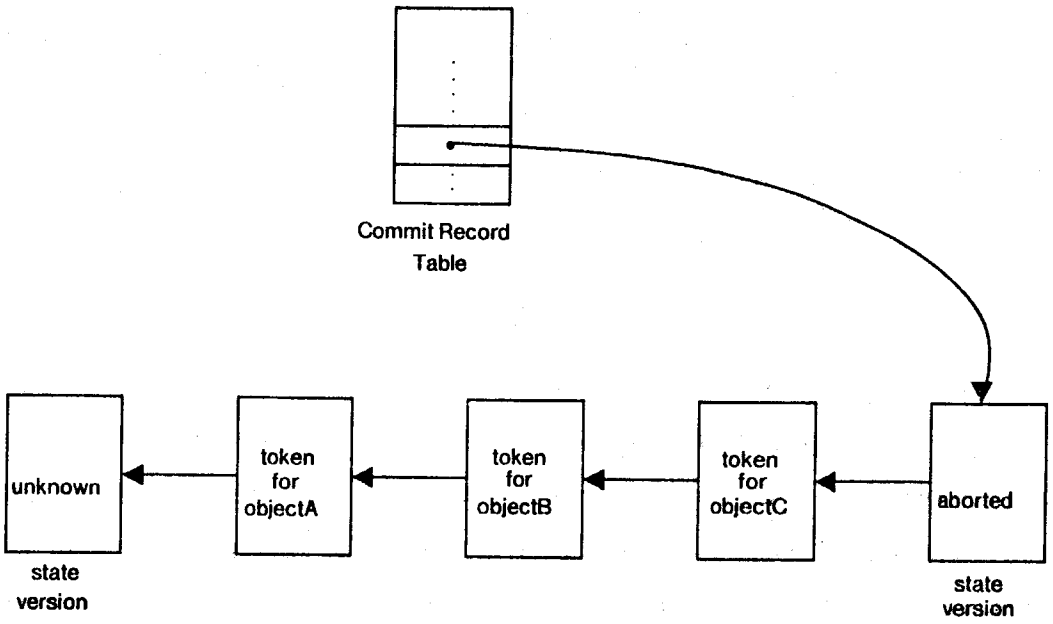


Figure 3-2: Structure of a Commit Record within the Repository

When the commit record is first created, an initial version is created. This version contains the unique identifier of the commit record, which is assigned by the repository, and the state of the atomic action, which is UNKNOWN. In addition, an entry (that points to this version) is created in the commit record table. Then, as tokens are created within the atomic action, they are not only threaded into the sequence of versions for their object, but are also threaded into the sequence of versions of the commit record. As each token is added to the commit record's list of versions, the corresponding commit record table entry is modified to refer to that token. Similarly, when a remote site adds a reference to the commit record, the repository creates a *representative* version, which contains the unique id's of the commit record and the remote site, and then threads that version into the commit record's sequence of versions. Finally, when the atomic action is committed or aborted, the repository creates another commit record version that contains the commit record's uid plus the final state.[5]

In order to carry out the final phase of the atomic action, in which all tokens are converted into versions or aborted form the object history, the repository modifies the object headers corresponding to each token in the commit record's sequence of versions, to reflect the final status of these tokens. The repository starts with the most current token in the list (which it accesses through the commit record table and then the first version which is the final state version) and when it reaches the initial state version of the commit record, it deletes the entry for that commit record from the commit record table.

3.3 Messages

The various types of messages that the repository can send and receive were listed in the table in Figure 2-4 in Chapter 2. Of these, all are single packet messages with the exception of create-token or version-value messages, which may contain large objects that cannot fit into a single packet. In these multiple packet messages, the sender places all of the information in the first packet, except for the fragments of the actual value of the object that

[5]Note that no object version will ever refer to a commit record that is created later than that object version. This invariant is used to optimize recovery, as will be seen in Chapter 4.

do not fit in this first packet. These fragments are the only data that will be contained in the subsequent packets. Figure 3-3 depicts both a multiple packet create-token message.

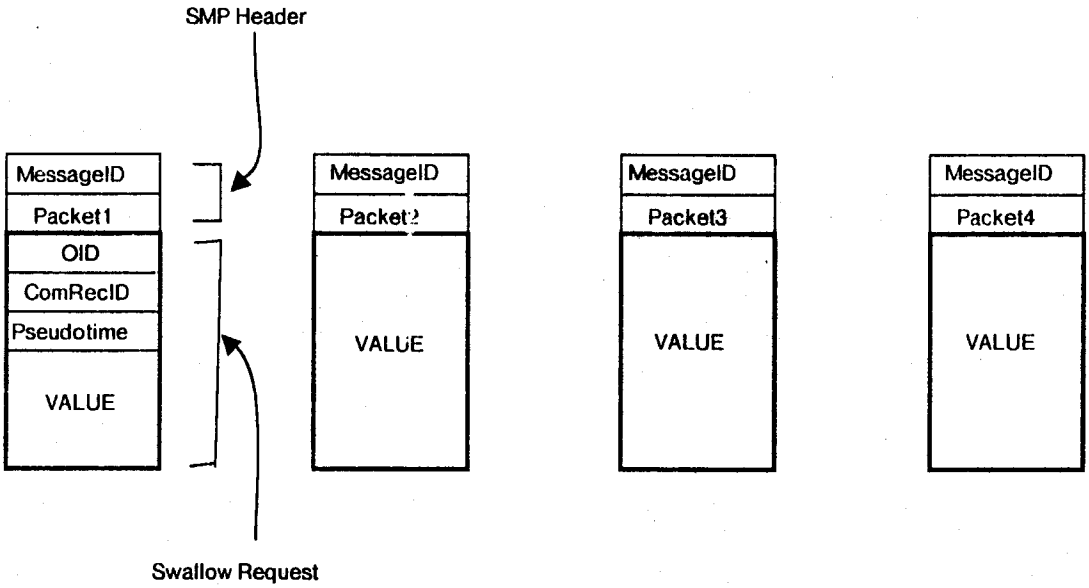


Figure 3-3: Structure of a Create-Token Message

Thus, when the repository receives a multiple packet create-token message, it does not have to wait for all of the packets to arrive before it can start writing the fragments of the object onto the disk. Instead, it can write the fragment contained within each packet as the packet arrives, and then can discard the packet since it has been processed.

3.4 Global State

There is a small amount of data that describes the repository's global state. Most of this data consists of the logical mappings of the various types of storage into the physical devices. The remaining data consists of values such as the last unique identifier that the repository assigned to an object or commit record, and data that describes certain recovery events. The nature of this data will become clearer by the end of the chapter.

3.5 Overview of Storage Organization

The repository supports several kinds of storage. Two are kinds of *atomic stable storage*, one is a kind of *careful storage*, and the remainder of the repository's storage is *volatile*. See Figure 3-4.

	Stable	Careful	Volatile
VS	X		
State	X		
Cache			X
OHS		X	
Temporary			X
Page Buffer			X

Figure 3-4: Storage Classification

Atomic stable storage, (henceforth referred to as stable storage), is secondary storage that we assume will never lose a value stored there. In practice, this means that stable storage contains multiple copies of these values at all times. These copies are organized so that it is unlikely that any one failure (such as a disk head crash) will destroy all copies of the same value. Furthermore, the repository's stable storage is *atomic* because a write to stable storage fails in only two ways - having made no change or having completed correctly. In general, the read and write operations on stable storage are time consuming since the multiple copies must be accessed and checked to be correct. The two types of stable storage in the repository are characterized as append-only and reusable stable storage. Append-only stable storage is like a tape since data is always written at the end. Also, no data is ever overwritten in append-only stable storage. On the other hand, in reusable stable storage, modifications made to the same data are rewritten in place.

Careful storage, is simply secondary storage in which there is only a single copy for each value stored there (not multiple copies as in stable storage). Thus, careful storage has faster

data access time than stable storage. Generally, the data in careful storage survives crashes, but it is not *guaranteed* to survive any crashes (as is guaranteed in stable storage). However, in the repository, the loss of data in careful storage does not cause failure as long as this loss can be detected, since the data can be recovered from the data in stable storage.

Finally, volatile storage is primary storage that is used as a temporary cache for the long term information stored in stable and careful storage. Volatile storage has a much faster access time than either type of secondary storage, but all data that it contains is lost when the repository crashes.

Thus, all data that is needed to represent the externally visible state of the repository is stored in stable storage so that if the repository crashes, none of this data will be lost. The versions of the objects and commit records are kept in append-only stable storage, called Version Storage and the global state data is kept in reusable stable storage, called State Storage.

The rest of the repository's data, which is redundant of information in stable storage or which does not have to be recovered at all after a crash, is kept in careful or volatile storage. Since the object header table would be too time consuming to recover in its entirety, it is kept in careful storage, called Object Header Storage. Then, if the repository crashes, only a small part of the table, if any, is lost. Thus, careful storage is used to improve the repository's performance by eliminating excessive accesses to stable storage while reducing the cost of recovery that would be required if the data were maintained in volatile storage. The commit record table, though, is smaller and less dense than the object header table, so it can be reconstructed much more easily after a crash. Therefore, it is only maintained in volatile storage. Finally, the messages that are pending when the repository crashes do not have to be recovered at all, since they are processed atomically and the protocols allow for incomplete mesages. Thus, message data is also kept only in volatile storage.

The remaining sections describe in detail the logical mappings of the repository's secondary storage (Version Storage, State Storage and Object Header Storage) into the physical devices as well as the methods used to encache in volatile storage the data that is

kept in secondary storage.

3.6 Version Storage

The main form of stable storage that the repository supports is Version Storage (VS) which contains the versions of objects and commit records as well as two other types of data, called checkpoint entries and epoch boundaries. (These checkpoint entries and epoch boundaries contain data that is used for recovery and will be described in Chapter 4). Abstractly, VS can be viewed as an infinite, append-only tape, but physically, it consists of 2 sets of write-once optical disks.[6] Each set is a backup for the other one in case some of the data is destroyed.

Since VS is append-only storage, it is always increasing in size. Thus, only a fraction of it can be kept on line. VS is managed in such a way that the current versions of objects and commit records remain in the portion of VS that is online. This online VS consists of the two or more most current disks of VS. The most current disk is called the *high space* and the oldest is called the *low space.* Online VS is managed as a circular buffer [SVOB80], as follows. When the high space is filled up, the current low space disk goes offline and a fresh disk becomes the new high space. Furthermore, whenever a version is accessed in the low space, it is copied into the high space. Thus, when the current low space disk goes offline, the version will still remain online.

All data is stored in VS in units called *version images.* There are 5 different types of version images: simple, root, fragment, boundary and checkpoint version images. A version image consists of size, type and data fields, and resides wholly within one page of VS. A version of an object or commit record that is small enough to fit on a single disk page is stored as a simple version image, as illustrated in Figure 3-5. However, a version that is larger than a single disk page needs a superstructure that points to all of the pieces of the version that are interspersed throughout several pages. Therefore a large version is stored as

---

[6] Initially, magnetic disks will be used to simulate optical disk. They will be used in a write-once manner, however.

a structure consisting of some number of fragment version images, which make up the version, plus a root version image, which has pointers to all of the fragments, as illustrated in Figure 3-5. A large version is written to the disk atomically by writing the root version image *after* all fragment version images are written and then, only linking the root into the appropriate sequence of versions. Thus, fragments of incomplete version images are ignored since they are unreachable. Finally, boundary and checkpoint version images look just like simple version images, except for the data field, which consists of an epoch boundary or checkpoint entry, respectively.

Several version images may be packed onto a VS page, which is the unit of physical reads and writes. In order to pack these version images as efficiently as possible, several unwritten VS pages are encached in a page buffer in volatile storage (recall that the disks are write-once only). Since VS is stable storage, it does not return the VS address of a version image (i.e., confirm the write to the repository process that initiated the write) until both copies of the VS page (on which the verison image resides) are written correctly from the buffer onto the two disks.

An unwritten VS page in the buffer is written out to the disks when either of the following three conditions holds true:

1. *The page is full* - Once a page is full, there is no need to wait any longer to write it out since it is only left unwritten in order to pack version images in it as tightly as possible.

2. *The page has been in the buffer for some extended period of time since the first version image was added to it* - Since a repository process cannot confirm external requests (that modify commit records or objects) until it receives a confirmation from VS and in turn, VS cannot confirm the write until the VS page is actually written on the disks, partially full pages are written out to the disks after a predefined time-out period. In this way, when the repository is not being heavily utilized, external requests will not remain unconfirmed for too long.

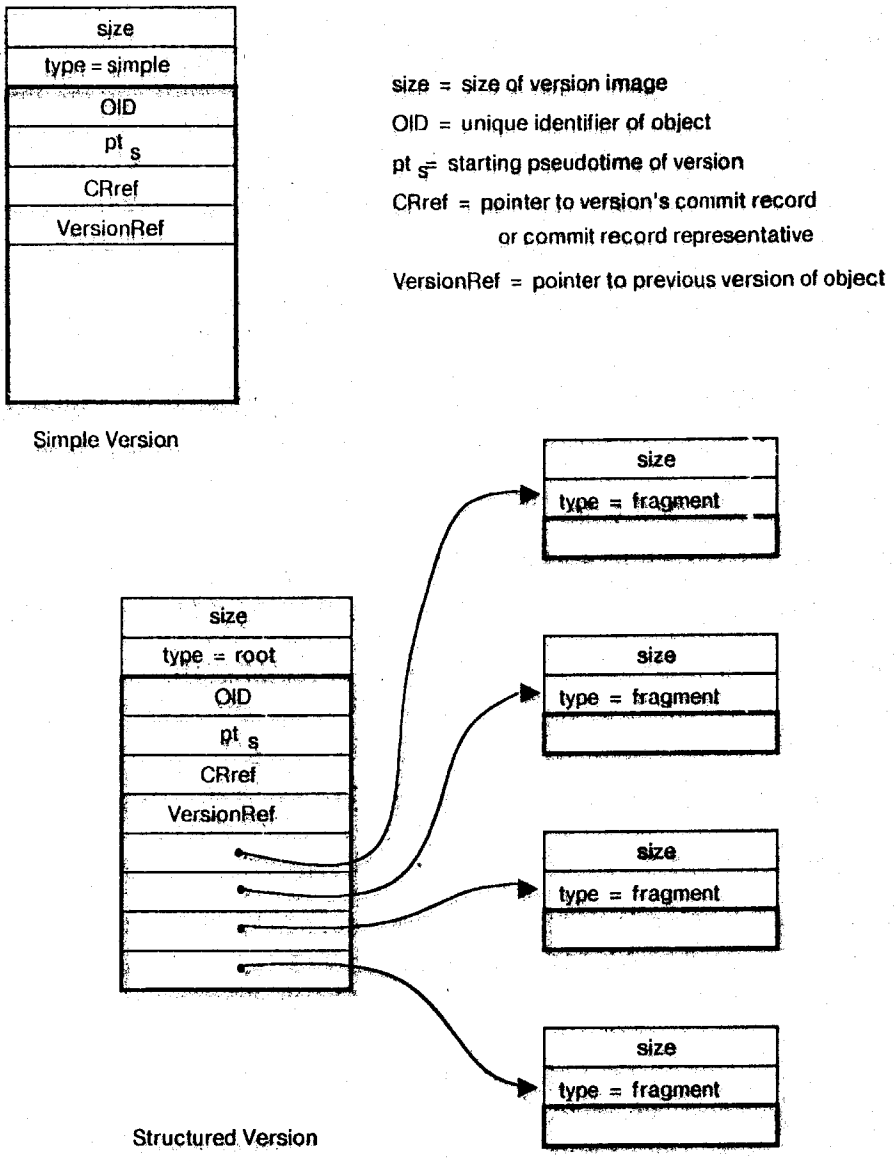


Figure 3-5: Simple and Structured Versions

Another unwritten VS page with a higher VS address is full and must be written out to the disks - This ensures that no version image is written at a lower VS address than any other version image to which it refers, and thus, preserves the abstract view of VS as an append-only tape. For example, in order for a process to create a version image, vi2, with a reference in it to another version image, vi1, it has to know the VS address of vi1. Since the process gets that VS address when VS confirms that vi1 has been written, then when the process requests that vi2 be written, all pages with VS addresses less than or equal to that of vi1 will already have been written on the disk and therefore vi2 cannot be written in any of them (VS disk pages are write once).

In order to actually write a VS page from the buffer, a copy is written to the same addressed page on each of two disks. After each copy is written out, it is read back to make sure that the correct data was written. Then, if a copy was not written correctly, it must be rewritten (and reread). However, it cannot be rewritten on the same disk page because the disk is write-once. Therefore, if either of the copies is written incorrectly, then both copies must be rewritten on another pair of pages.

In addition to maintaining several unwritten pages in the buffer, several of the most recently written or read VS pages are also encached in this page buffer so that if these encached pages are read again within a short time period, the disks will not have to be accessed. However, if a process wishes to read a version image on a page that is not in the buffer, then the disks have to be accessed, as follows. First, one copy is read from the disk and verified to be correct, using a checksum. If that copy is correct, then the second copy does not have to be examined. On the other hand, if that copy is incorrect then it must be recovered from the second copy.

In order to implement this recovery, both copies of the page must be rewritten on a new set of identical disk pages, as is done when the write operation fails. However, all references to the version images on a page that has been recovered in this way would become invalid. Thus, in order to preserve the validity of these references, the repository maintains a map from the bad pages to their replacement pages. Then, when a process attempts to access a version image on a bad page, VS will find the recovered copies of that page, using this map.

Once a page is determined to be bad, it should never be mistaken for a good page. Thus,

the page must be made detectably bad forever. If VS is implemented using optical disks, as originally planned, then pages can be made bad permanently by writing on them a second time, obliterating any marginal data. However, if another type of disk is used, then some other method, such as keeping a table of bad pages, would have to be devised in order to make pages detectably bad forever.

3.7 State Storage

The second form of stable storage that the repository supports is called State Storage, which contains the data that describes the repository's global state. Physically, state storage consists of a small amount of reusable magnetic disk storage. It is stable due to the fact that the global state data is duplicated at separate locations on disk that have independent probabilities of decaying. In other words, it is not probable that a single crash can destroy both copies of the data.[7]

The repository supports State Storage in addition to VS for the combined reasons that the location of the global state cannot change and VS is write-once only. If the global state was kept in VS, then every time it was modified it would be written into a new location in VS. This would mean that when the repository was booting itself after a crash, it would not know exactly where to find this data because its location could not be hardwired into the bootstrapping procedure. By supporting reusable stable storage, this problem is avoided.

In order to write a State Storage page, each of the copies is written and then read back (to verify that the copy was written correctly). However, since writing a State Storage page overwrites older copies of the global state, the copies must be written and read back sequentially instead of in parallel, as in VS. Then, if the repository crashes in the midst of writing one copy, there will still be another valid copy from which to recover the data that is contained on that State Storage page. Furthermore, the copies are always written in the same order so that if a failure occurs in between writing the two copies (leaving both copies

[7] In order to be even more reliable, the actual implementation of State Storage may keep 3 copies of all data.

valid but different), the repository will know which copy is current.

In order to read a State Storage page, both copies on the disk must be verified to be correct and identical to one another before allowing any repository processes to examine the page. If either one is bad, then the bad copy is recovered from the good copy. Further, if both copies are valid but not identical, then the second copy is recovered from the first copy, which is the current copy. It is not sufficient to verify the correctness of only one copy, when reading a State Storage page from the disk, because the repository may have previously crashed before writing the second copy. If the second copy is not subsequently updated when read, then another write of that State Storage page could fail and damage the first copy, leaving no valid copy from which to recover. (The second copy would be too far out of date to be of any use). Thus, when reading a State Storage page, it is necessary to compare both copies and recover one, if necessary.

Since the global state data is read fairly frequently, it is encached in volatile storage to eliminate the time consuming accesses to State Storage. Thus, the only time the disk has to be accessed in order to read the global state is when the repository first comes into existence, and then, whenever the repository restarts after a crash. On the other hand, since most of the State Storage data changes fairly infrequently, if at all, it is kept current in State Storage (that is, every time it is updated in the cache it is also written onto the disk). There are two values, though, that change too often to be practically kept up to date in State Storage. Thus, they are kept current in the cache, but are only periodically updated in State Storage. These two values are the VS write pointer, which indicates the current end of VS and the value of the last unique identifier that the repository assigned to an object or commit record. The write pointer is only updated in State Storage every N^{th} time its value changes, where N is a predefined constant. Similarly, the value of the last uid (unique identifier) assigned is only updated in State Storage every X^{th} time its value changes, where X is another predefined constant. The recovery of these two values after a crash will be described in the next chapter.

3.8 Object Header Storage

Object Header Storage, or OHS, is reusable careful storage in which the object header table is maintained. The repository keeps this table of object headers so that it does not have to scan sequentially through VS in order to find the versions of objects. An object header provides direct mappings to the current version and token as well as a reference to the token's commit record.

Even though object headers are not required in order for the repository to function *correctly* (the repository can always resort to a sequential search through VS) they are necessary in order for the repository to function *efficiently*. Therefore, the object header table must be organized so that the object headers are efficiently accessible. The two main alternatives for the table structure were a B-tree or a hash table. A non-coalesced chain hash table similar to the one described in [7] was selected.

This type of hash table was chosen for its simplicity of structure and ease of recovery, as well as for its efficient search, insertion and deletion algorithms. The average search time of the hash table is independent of the size of the table (providing that the table does not get too full) while the average search time of a B-tree is directly proportional to the logarithm of the table size [7, 4]. Also, the fundamental unit of a linked list in the hash table (a bucket) contains only a single object header, whereas that of a linked list in the B-tree (a node) usually contains some number greater than one. Therefore, there is potential for losing more information in a B-tree than in a hash table if a link is broken (e.g., when one of the fundamental units gets lost or becomes obsolete after a crash). Finally, it is easier to characterize the problems that can arise in the hash table as a result of a crash than in a B-tree. Therefore, the hash table was more easily adaptable to recovering itself in the background as the repository fulfills requests.

The basic structure of the hash table is as follows. The OHS pages are divided into fixed size units, each of which can accommodate a single object header. Each of these units is a bucket in the hash table and is uniquely identified by its OHS address. Further, only three of the object header fields are relevant to the hash table: the OID, the delete flag and the hash table link. The OID is used as a key in the hash table. Thus, a mathematical function

is used to map or *hash* every OID to some bucket in the table. The bucket to which an object header hashes will be referred to as its *home* bucket. Next, the delete flag is used to indicate whether the object header is valid or has been deleted. Finally, the hash table link is used to create linked lists of buckets. The remaining fields of the object header are ignored by the hash table algorithms.

Even though only one object header can occupy a bucket at any given time, there exists more than one object header (or more specifically, OID of an object header) that hashes to each bucket in the table. Therefore, once a bucket is occupied, all other object headers that are added to the table, whose home bucket is that bucket, are placed arbitrarily in other empty buckets and linked together.[8] The first bucket in each linked list is the one to which all of the object headers in the other buckets hash, i.e., it is their home bucket. The linked lists will be referred to as *chains*.

Figure 3-6 illustrates a page in the hash table to be used in examples throughout this thesis. All figures that depict pages of the hash table will be of the same form but will show only the contents of the pages and buckets that are relevant to the particular example. There are four pages, A through D, in the hash table, each containing five buckets. The object headers have OID's of the form ohN, where N is the OID (an integer). Chains are identified by the address of the home bucket. Also, the two states of the delete flag will be represented by the letters V (valid) and D (deleted). The remaining fields within the actual object header are not relevant to the discussion about the organization of OHS, so they will simply be represented in each bucket as an X mark. Finally, the hash function selected for the examples in this thesis is OID modulo 20.

The three hash table operations are search, insertion, and deletion. The search operation finds the specified object header in the object header table. The insertion operation is used for adding newly created object headers to the object header table. Finally, the deletion operation simply eliminates an object header from the object header table.

---

[8]The choice of buckets is not completely arbitrary. The algorithm for finding a free bucket first looks for a bucket on the same page as the home bucket since in this way, most linked lists will be constructed so that they are fully contained within a page and thus, the amount of paging that must be done will be minimized.

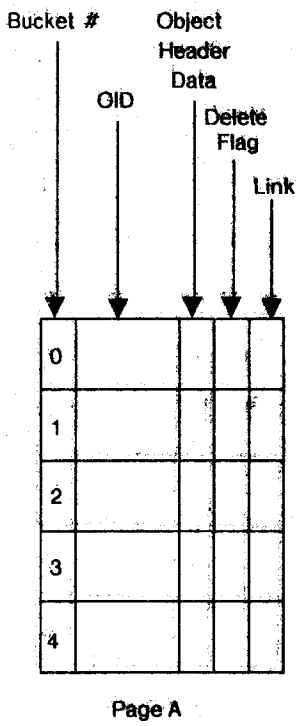


Figure 3-6: A Representative Hash Table Page

The search algorithm is as follows:

1. Hash the given object header to the home bucket, X.

2. If bucket X is empty or contains an object header whose home bucket is not bucket X (i.e., hashes to another bucket), then terminate unsuccessfully. Otherwise continue searching down chain X until the requested object header is found or the end of the chain is reached.

3. If the end of the chain is reached then terminate unsuccessfully. Otherwise, return the object header that was found.

An example:

Suppose pages C and D of the hash table are as shown in Figure 3-7 and we wish to find oh37. Oh37 hashes to bucket 17, so we first check to see if bucket 17 contains oh37. Since it does not, we hash the object header in bucket 17 to see whether bucket 17 is the home bucket for that object header. Since it is, we follow the links through successive buckets in chain 17 and find oh37 in bucket 15.

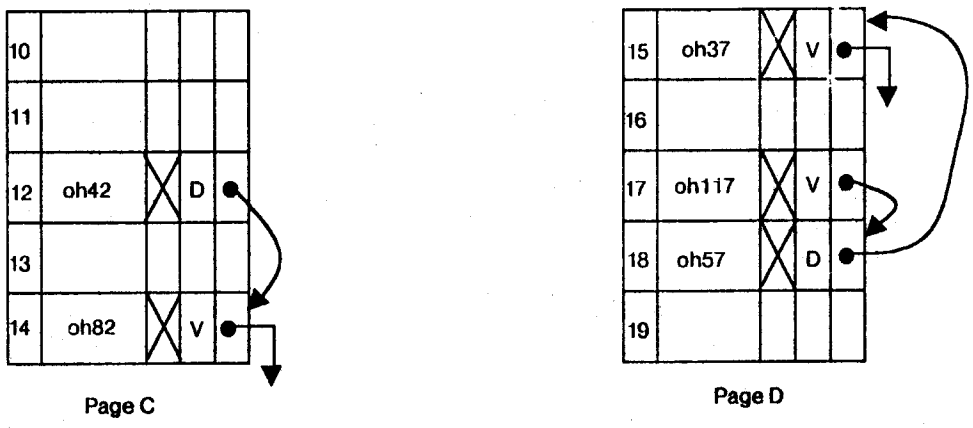


Figure 3-7: Initial State of Pages C and D

The insertion algorithm is as follows:

1. Perform the search operation on the object header.

2. If the search terminates successfully, finding an older version of the object header in bucket B, then insert the updated version of the object header in B and terminate. Otherwise, hash the object header to the home bucket, X.

3. Do one of the following:

   a. If bucket X is empty, or contains a deleted object header whose home bucket is bucket X, then simply insert the new object header into bucket X.

   b. If bucket X contains a valid object header whose home bucket is bucket X, then check for another bucket on chain X that contains a deleted object header. If one exists then insert the object header there. Otherwise, find another available bucket, Y, insert the object header in it, and add it to the end of chain X.

   c. If bucket X contains an object header whose home bucket is not bucket X, then bucket X must be part of another chain, beginning with bucket Z. Thus, it is necessary to move the object header presently in bucket X to some other bucket. If there is a bucket, D, on chain Z that contains a deleted object header, then move the object header in bucket X to bucket D. Otherwise move the object header in bucket X to a free bucket, F, and reroute chain Z through bucket F. Once the old object header has been removed from bucket X, insert the new object header there.

The following is an example of three successive insertions that are executed on the hash table shown in Figure 3-7. Each insertion demonstrates one of the branches that can be taken in Step 3 of the insertion algorithm.

Suppose we wish to insert oh12 into the hash table. We perform the search through chain12 in page C (Figure 3-7) and it terminates unsuccessfully. Next we check the object header (oh42) in bucket 12 and discover that bucket 12 is its home bucket but it is marked deleted. Therefore we execute step 3a of the insert algorithm by discarding oh42 and inserting oh12 in its place in bucket 12. See Figure 3-8 for the state of page C after this insertion is done.

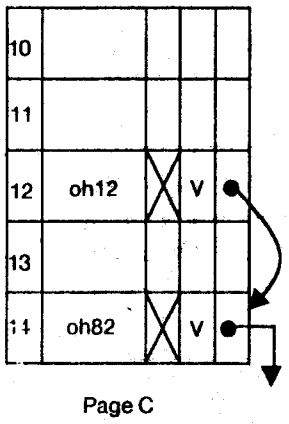


Figure 3-8: Page C After Oh12 is Inserted

Now, suppose we wish to insert oh77 into the hash table. We cannot place it in bucket 17 in page D (Figure 3-7) because it is the home bucket for the object header that it contains and that object header is still valid. Therefore we look for another bucket already on the chain that contains a deleted object header. Bucket 18 satisfies these requirements so we execute step 3b of the insertion algorithm and insert oh77 in bucket 18 in place of oh57. Figure 3-9 shows what page D of the hash table looks like after this insertion is done.

Finally, supppose we wish to insert oh34 into the hash table. We have to execute step 3c of the insertion algorithm because bucket 14 in page C (Figure 3-8) is not the home bucket for the object header that it contains, oh82. Therefore, we move oh82 to another free bucket, bucket 10, then reroute chain 12 through bucket 10 and finally, insert oh34 into bucket 14. See Figure 3-10 for the final state of page C after this insertion is done.

The deletion algorithm is as follows:

1. Perform the search operation on the object header.

2. If the search terminates unsuccessfully (i.e. the object header is not found) then terminate unsuccessfully. Otherwise change the state of the bucket in which the object header was found to *deleted.*

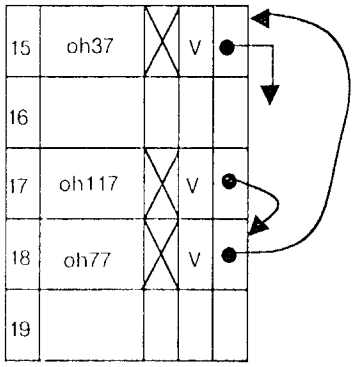


Figure 3-9: Page D After Oh77 is Inserted

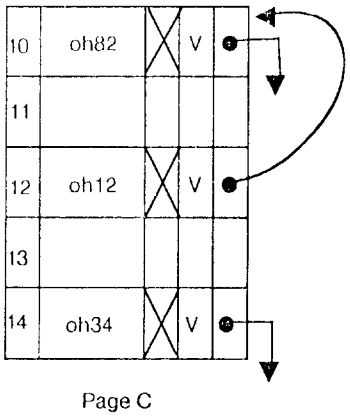


Figure 3-10: Page C After Oh34 is Inserted

An example (using the hashtable shown in Figure 3-9):

Suppose we want to delete oh37. We find it in bucket 15 and simply mark it *deleted* as shown in Figure 3-11.

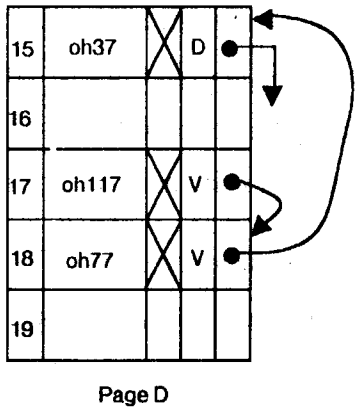


Figure 3-11: Page D After Oh37 is Deleted

When an object header is deleted it is not removed from the bucket in which it resides nor is the bucket removed from the chain of which it is a part. These actions are delayed until some time in the future when another object header has to be inserted and an empty bucket is needed. Then, if the deleted bucket is part of the chain to which the object header to be inserted belongs, the object header can be inserted into the bucket in place of the deleted object header without making any changes to the chain structure. (This was the case in the first two examples of insertions). This eliminates the work involved in restructuring the chain, for both the deletion and insertion algorithms. At worst, if a bucket is needed to hold another object header that does not belong in the chain of which the bucket is a part, then the restructuring has to be done anyhow.

The deletion algorithm delays the actual removal of the object header from object header table in order to alleviate the following problem. Since pseudotimes do not directly correspond to real time, read requests for an object may arrive after that object has been deleted with respect to real time but before the object has been deleted with respect to

pseudotime. Thus, it is hoped that in most cases where this situation arises, by delaying the actual removal of the object header from OHS, the object header will still be available so that the repository does not have to scan sequentially through VS in order to find the appropriate version.

In OHS, like in VS, the fundamental unit of read and write is actually a page. Also, several of the most recently read and written OHS pages are encached in the page buffer in volatile storage. However, unlike in VS, an object header does not have to be written from the page buffer to the disk before a repository process can confirm an external request, since data may get damaged even if it has been written on the disk (OHS is not stable storage).

Furthermore, the object header table may not be modified atomically, since the insertion algorithm sometimes modifies object headers on several pages, which are not written to the disk in any related order nor all at once. The object header table is not modified atomically because many independent processes may be concurrently inserting object headers on the pages in the buffer and thus, there may be no instant in time (except for when the repository is idle) when all of the object headers on a page or set of pages are consistent and hence, atomically writeable.

Therefore, a page that has been modified in the buffer is actually written out from the buffer to the disk when one of the following conditions holds true:

1. *The page is the least-recently-used page in the buffer and another page has to be brought into the buffer* - The OHS page buffer replacement scheme is a Least-Recently-Used scheme.

2. *An extended period of time has passed since the page was modified in the buffer* - This prevents pages that are frequently being accessed from getting too obsolete on the disk.

3. *The repository has no more outstanding requests* - At this time, all pages in the buffer that haven't been written to the disk since they were last modified, are written. This brings OHS to a consistent state.

However, it would be very rare for the repository to crash in the midst of a non-atomic insertion operation, for the following reasons. First, the insertion algorithm is only executed when object headers are initially created. Whenever they are modified, the repository

process requesting the modification would have obtained the OHS address of the object header when it read that object header. Thus, unless the object header was moved, the insertion operation wouldn't have to be executed since the object header could be modified directly, using the OHS address. Second, most chains are completely contained within a single page, so even if the insertion algorithms modifies several buckets on the chain, the object header table will still be updated atomically (each page is written atomically).

Thus, in the few cases where a crash causes the object header table to be updated non-atomically, the repository's recovery mechanisms will restore consistency within the object header table. This, and all other recovery will be described in the next chapter.

Chapter Four

Recovery of the Repository

In order to recover from a crash, the repository must restore its global state, as well as the state of the objects and commit records, to a state that is current with respect to that of Swallow as a whole. On the other hand, the repository does not have to recover the messages that were left pending when it crashed, for reasons that will be described in this chapter.

Since some of the global state data consists of recovery information that has not yet been described, the discussion of the global state data's recovery will be deferred until Section 4.3, at which point the recovery information will have been described. But first, Sections 4.1 and 4.2, respectively, discuss how the internal structure of the objects can be damaged by a crash and also describe the individual recovery mechanisms that are used to implement their recovery. Then, Section 4.3, presents the recovery manager, which coordinates all recovery activities. This section explains how the global state data is recovered as well as how the various recovery mechanisms are integrated into a coherent recovery process that interfaces with the processes that are satisfying external requests, concurrently. Finally, Section 4.4 explains why it is unnecessary to recover the pending messages.

4.1 Recovery of Objects

Due to the fact that VS is stable storage, and thus, maintains all of its data redundantly, all object versions that are confirmed to have been written there will be found there after a crash. Furthermore, all incomplete versions are ignored. Thus VS, in itself, contains the current state of all objects. Were it not for a desire to improve performance, elaborate recovery mechanisms would not have been needed. However, to find the most current version of an object in VS requires a linear search, which would perform very poorly. To

overcome this performance problem, the repository accesses the objects' versions in VS through the object header table, which is maintained in OHS. Since OHS is only careful storage, a crash may damage the structure and/or contents of the object header table. Thus, it is this object header table that must be recovered in order for the objects to be consistent with the general state of Swallow.[9]

The various types of structural damage to which OHS is vulnerable are merged, cyclic and incomplete chains (Section 4.1.1). The repository uses a modified set of hash table algorithms (Section 4.1.2) in order to detect and correct these damaged chains. On the other hand, the contents of the object header table, that is the actual object headers, get damaged by becoming lost or obsolete (Section 4.1.3). Most of the information contained in these lost and obsolete object headers can be recovered from the data in VS, as described in Section 4.1.4. Furthermore, the repository uses two mechanisms, *recovery epochs* and *checkpoint epochs* (Sections 4.1.5 and 4.1.6, respectively), in order to facilitate the recovery of these lost and obsolete object headers.

4.1.1 Merged and Cyclic Hash Table Chains

When an object header is inserted into the object header table, several buckets may be modified. If these buckets are not all located on the same disk page then all of these modifications may not be atomic, since the OHS page buffer management scheme does not write the separate pages out to the disk in any particular order nor all at once. If it was possible to write out the pages so that each bucket is written out before any other buckets closer to the end of the chain then all problems except for incomplete chains would go away. However, since many processes may be concurrently accessing buckets on different chains but on the same OHS pages, it may not be possible to preserve any such order. Furthermore, since the cost of the OHS operations (and thus, the repository's response

---

[9]Note that implementing OHS as atomic stable storage would not really alleviate this problem. The lost object header problem would go away but there would still be a problem of inconsistency between OHS and VS, since every object history operation involves touching both. The cost of this alternative is discussed in Chapter 5.

time) would increase if the concurrency of accesses to buckets on a single OHS page was eliminated. the OHS page buffer does not ensure atomicity of insertions of object headers into OHS. This non-atomic insertion of object headers is manifested after a crash in one of three types of malformed hash table chains: merged, cyclic, or incomplete.

Merged Chains:

A chain is considered to be *merged* when its last bucket contains a link to a bucket that is part of another chain. In Figure 4-1, chain 1 is merged with chain 5. One way in which chain 1 could have become merged with chain 5 is as follows.

Assume that the initial state of pages, in both the buffer and on the disk, is as illustrated in Figure 4-2 and that oh5 is to be inserted. In order to insert oh5, oh101 would have to be moved to another empty bucket and chain1 would have to be rerouted through the new bucket. Figure 4-3 show how pages A and B would appear in the page buffer after the insert was done. However, if the repository crashed before page A was written on the disk but after page B was written, then chain 1 would merge with chain 5 as originally illustrated in Figure 4-1.

Since merged chains are longer than necessary, they tend to reduce the efficiency of the hash table algorithms. Furthermore, if a merged chain is not corrected before subsequent operations modify it, then it may become merged with an additional chain each time the repository crashes, forming a single long chain. Thus, when merged chains are not corrected, the original benefit of a hash table is lost, since the efficiency of the algorithms is reduced.

In addition, the longer the repository waits, the more difficult it becomes to fix a merged chain. It is easy to fix a chain when it initially becomes merged because all of the buckets from one chain are located at one end of the merged chain and those from the other chain are located at the other end. Thus, only one link has to be modified in order to correct the situation. However, as additional insertions and deletions are executed on the merged chain, the buckets of the two component chains become interleaved, as shown in Figure 4-4. Thus, it would be necessary to break several links and then relink the buckets properly in order to reconstruct two separate chains.

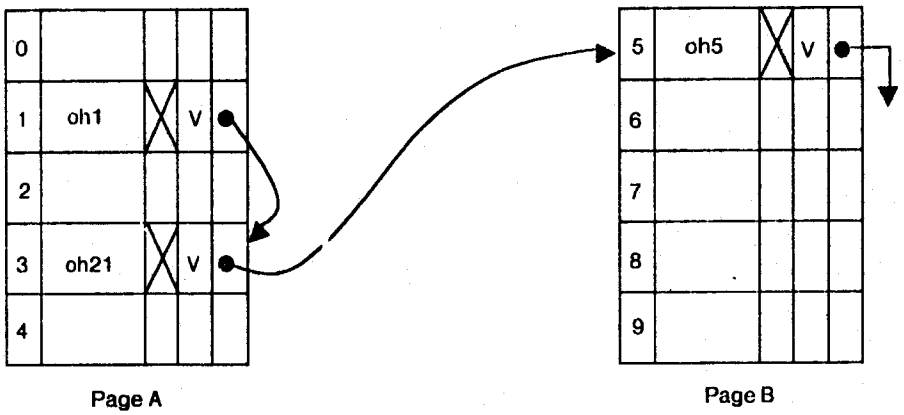


Figure 4-1: A Merged Chain

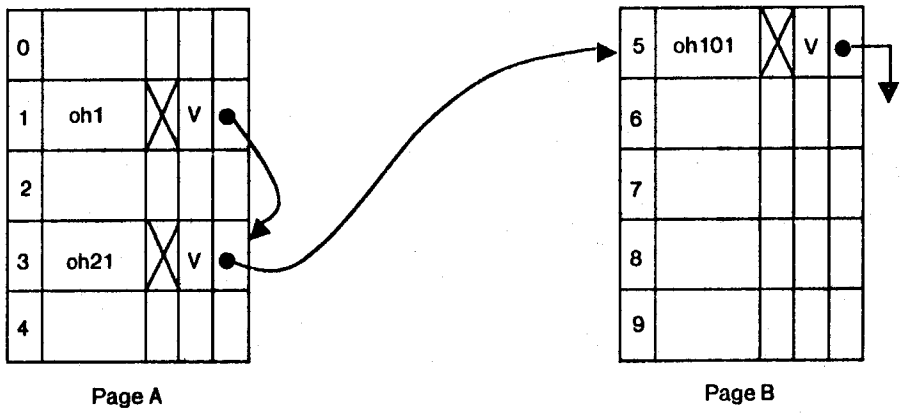


Figure 4-2: Pages A and B Before Insertion of Oh5

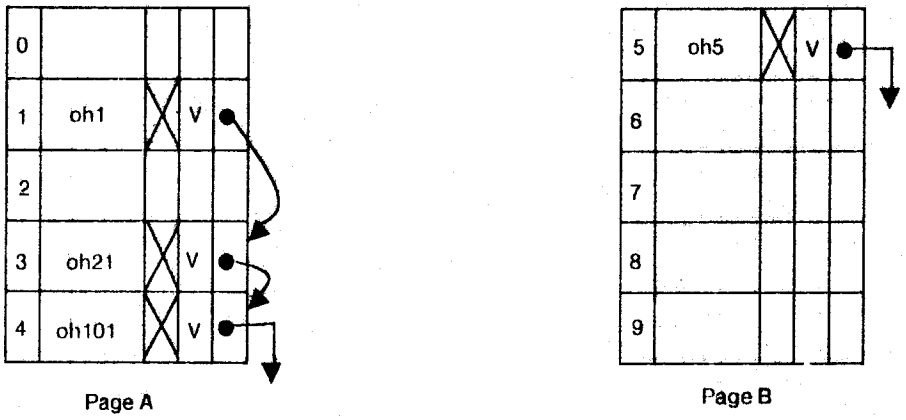


Figure 4-3: Correct Insertion of Oh5

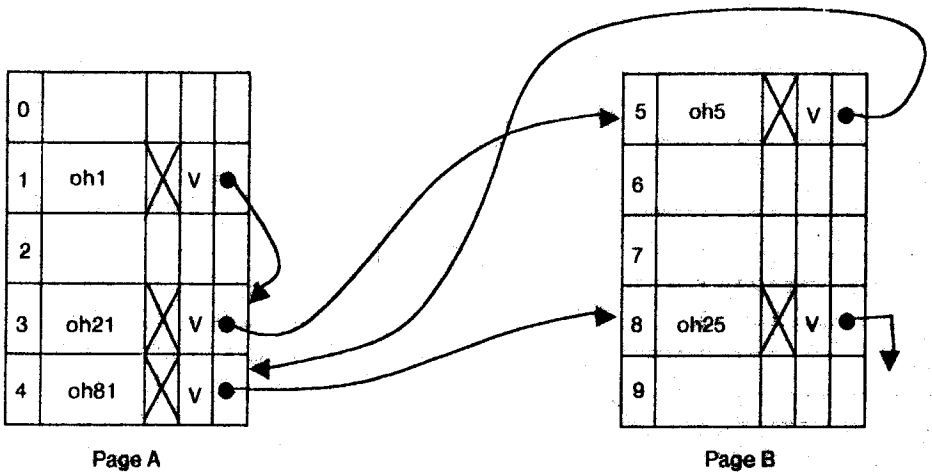


Figure 4-4: Merged Chain with Interleaved Buckets

Cyclic Chains:

A *cyclic* chain contains a bucket whose link points back to another bucket (also in that chain) that is closer to the beginning of that chain, as illustrated in Figure 4-5. Cyclic chains are undesirable because they prevent the hash table algorithms from terminating. In other words, these algorithms become infinite loops when executing on cyclic chains because they never encounter a null chain link, which signals that the end of the chain has been reached.

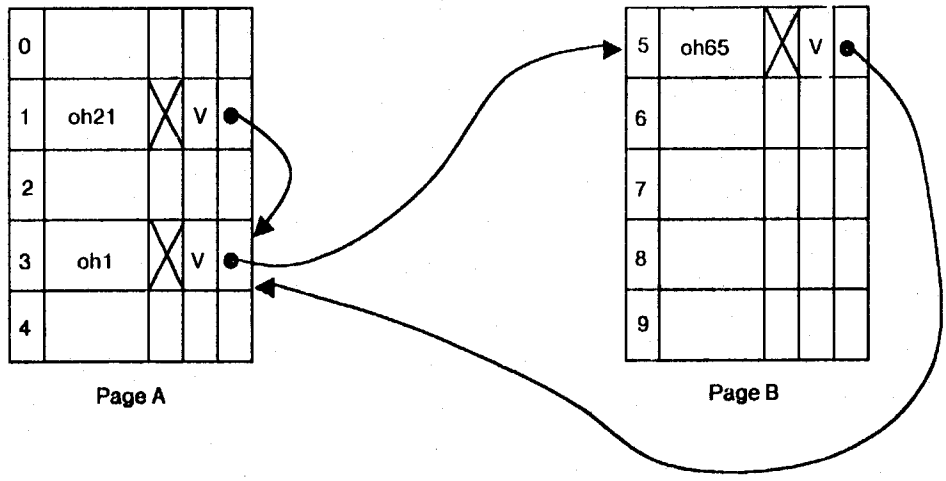


Figure 4-5: A Cyclic Chain

For example, assume that the state of pages A and B in the buffer and in OHS is as shown in Figure 4-6, and that the following sequence of operations is executed.

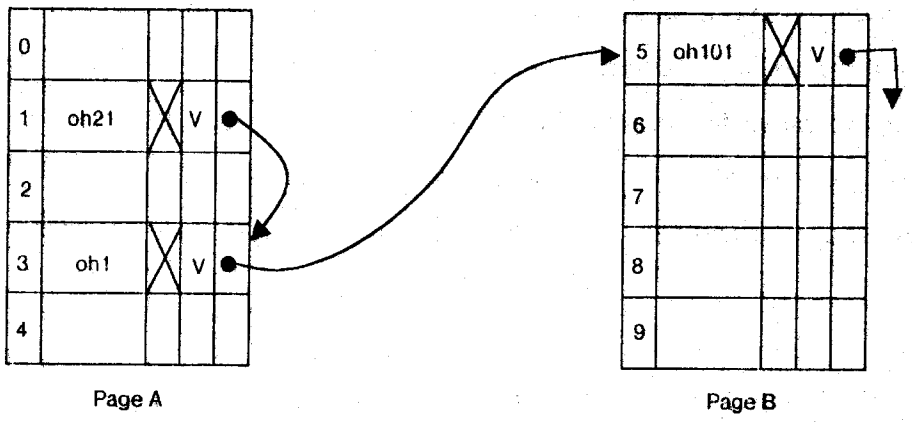


Figure 4-6: Pages A and B Before Cycle was Created

1. Oh1 is deleted from chain1.

2. Oh101 is deleted from chain1.

3. Oh65 is inserted in chain 5. (In doing so, a collision occurs in bucket5. Normally, oh101 would have to be moved to another bucket and chain 1 rerouted through it, but since oh1 was deleted, this is not necessary. Thus, oh101 is simply removed from the table, oh65 is inserted in bucket 5 and chain 1 is modified so that it no longer includes bucket 5.)

4. 0h105 is inserted in chain 5. (Again, a collision occurs but this time with an object header that belongs on the chain. Therefore, oh105 has to be inserted into another free bucket. Assuming that bucket 3 is the bucket that is found to be free [oh1 is deleted and therefore can be removed from the bucket], oh105 is then inserted in bucket 3, which is then added to chain 5.)

Figures 4-7, 4-8, 4-9, and 4-10 show the pages as they would appear in the buffer after each step is executed. Now, if a crash occurs at a point when page A (on the disk) is still in the same state as before any of these operations were executed yet page B has been written out to the disk in the final state, then chain 1 becomes merged with chain 5, and the resulting chain contains a cycle, as previously shown in Figure 4-5.

When a cycle is initially created in a chain, it is always accompanied by the merging of

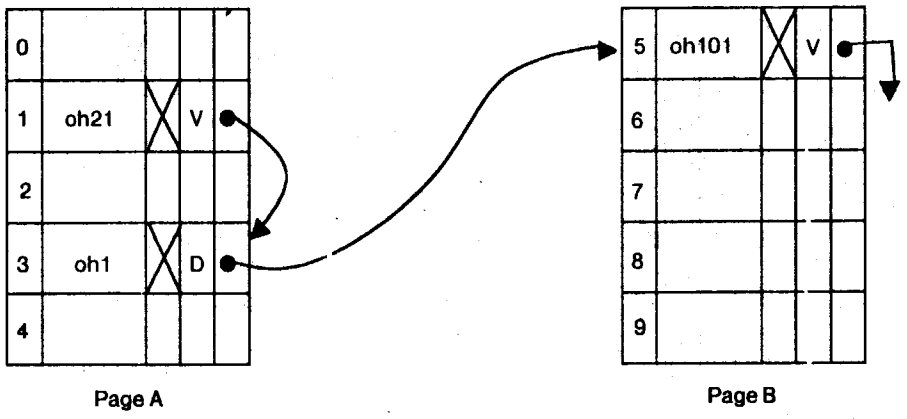


Figure 4-7: Deletion of Oh1

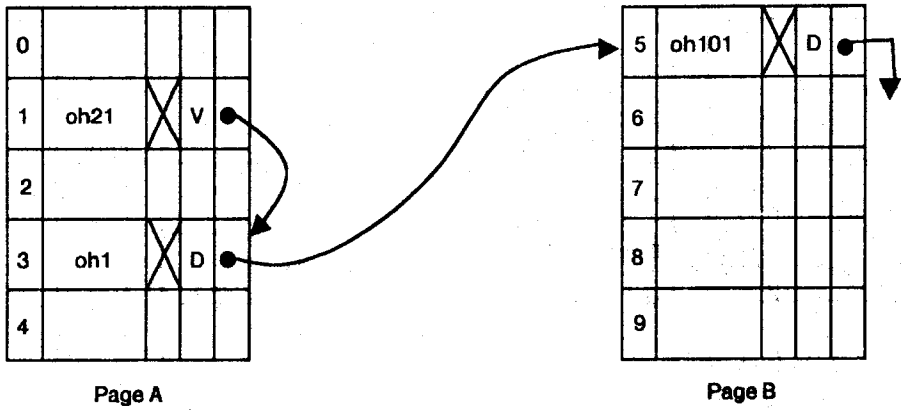


Figure 4-8: Deletion of Oh101

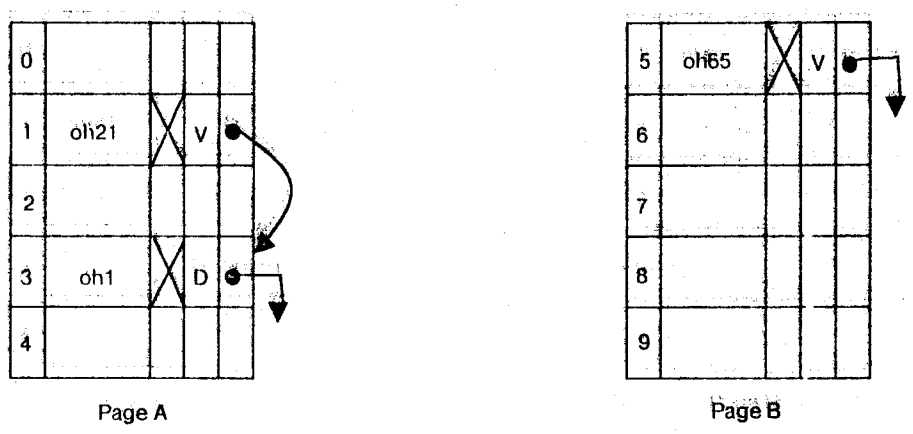


Figure 4-9: Insertion of Oh65

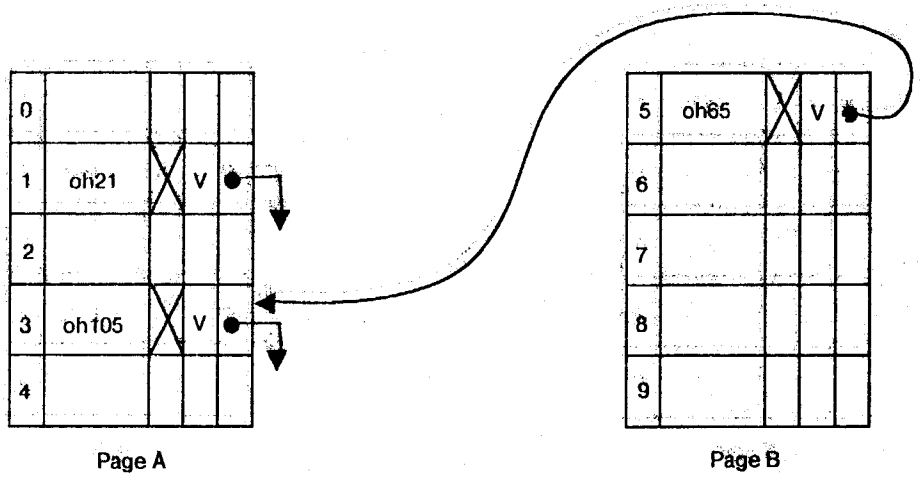


Figure 4-10: Insertion of Oh105

two chains, as demonstrated in the previous example. Therefore it would seem that in order to detect a cyclic chain, one would simply check for a merged chain. However, this detection procedure would not catch all cyclic chains if they were not always corrected before allowing subsequent operations to modify them. For example, suppose that oh27 is to be inserted in the cylic chain illustrated in Figure 4-11. Since bucket 7 is not oh81's home bucket but is oh27's home bucket, oh81 must be removed from bucket 7 and oh27 must be inserted in oh81's place. Furthermore, if possible, oh81 should be moved to another bucket on chain 1 that contains a deleted object header. Since bucket 12 is on chain 1 and contains a deleted object header, oh81 is inserted there after the deleted oh72 is removed. Finally, oh27 is inserted in bucket 7 and chain 1 is rerouted around it. The final state of pages A, B, and C is shown in Figure 4-12. Since there is no merged chain anymore, the cycle would not be detected by the simple detection procedure that was proposed above. Thus, as is the case with merged chains, it is advantageous to correct the damage in cyclic chains before allowing further operations to modify it.

Incomplete Chains:

An *incomplete* chain is one in which the tail end of the chain is unaccessible, that is, the last reachable bucket in the chain contains a pointer to an empty bucket or a bucket on a damaged page.[10] For example, one way in which an incomplete chain could be created is as follows. Assume that the initial state of the pages is as shown in Figure 4-13 and that oh81 is to be inserted. In order to insert oh81, it is necessary to find a free bucket, insert oh81 in it, and then add the bucket to chain 1. However, if the only free bucket is bucket 5 and the repository crashes before writing page B but after writing page A in OHS, then the chain becomes incomplete, as shown in Figure 4-14.

[10]Thus, incomplete chains are caused not only by non-atomic insertions of object headers, but also by bad OHS pages.

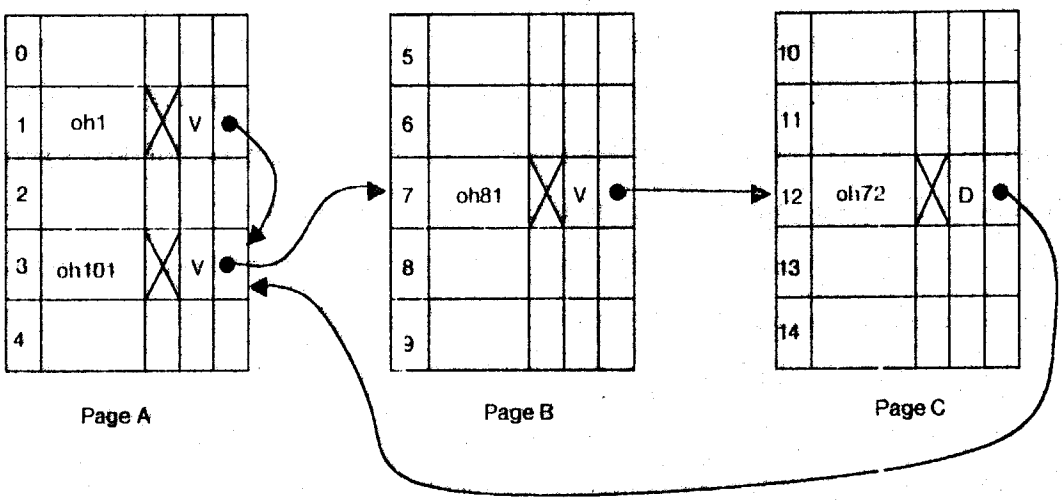


Figure 4-11: Pages A, B and C Before Oh27 is Inserted

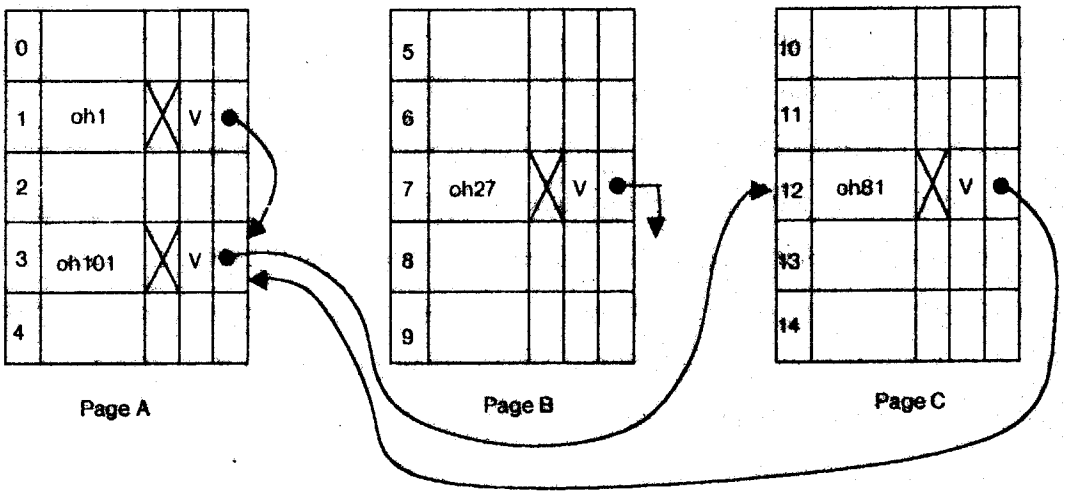


Figure 4-12: Pages A, B and C After Oh27 is Inserted

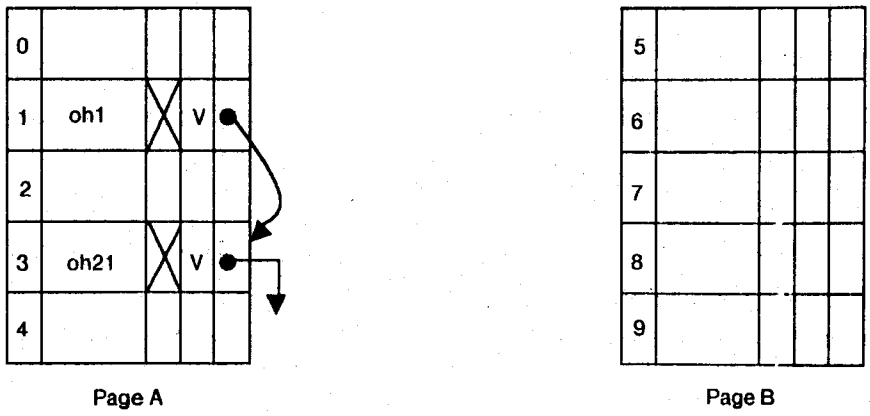


Figure 4-13: Pages A and B Before Oh81 is Inserted

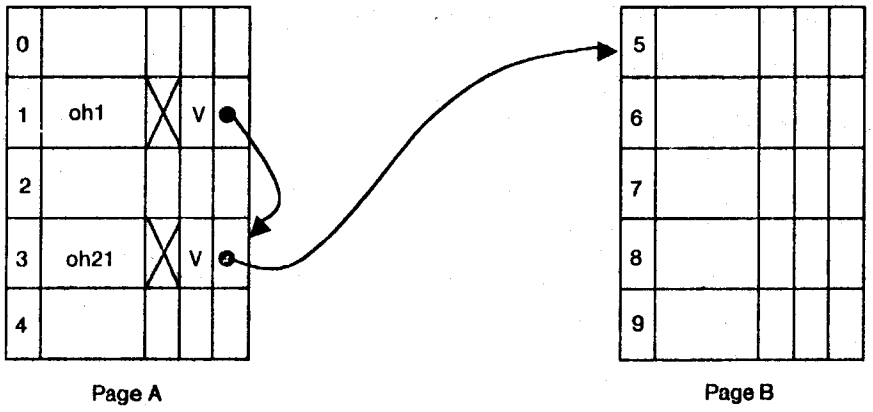


Figure 4-14: Pages A and B After Crash

4.1.2 A Modified Set of Hash Table Algorithms

Here we describe simple modifications to the insertion, deletion and search algorithms that make the hash table self-recovering with respect to the structural damage that has just been described. First, straightforward consistency checks are incorporated into the algorithms in order to detect defects in a chain before any operations modify the chain. Then, if a defect is discovered, a simple correction procedure is applied in order to return the chain to a state in which it can be safely operated on.

In order to simplify the explanations of the modified algorithms, a *defective* bucket is defined to mean a bucket with one of the following properties:

1. The bucket is supposed to contain an object header but instead, is empty.

2. The bucket contains an object header whose home bucket is not the first bucket of the chain to which it is linked, and therefore, does not belong in that chain.

3. The bucket is located on a bad OHS page, and thus, cannot be accessed.

In the modified algorithms, every chain that is touched is checked to ensure that none of its buckets are defective. If a defective bucket is found, then the link of the preceding bucket (which points to the defective bucket) is changed to nil, thereby separating any merged chains, breaking any cycles before the hash table algorithms become trapped in them, and repairing the improper link in any incomplete chains.

More specifically, the modified search algorithm is as follows (note that all of the changes and additions are italicized):

1. Hash the given object header to the home bucket, X.

2. If bucket X is empty or contains an object header whose home bucket is not bucket X, then terminate unsuccessfully. Otherwise, continue searching through chain X until either the object header in question is found , *a defective bucket is found*, or until the end of the chain is reached.

3. If the end of the chain is reached then terminate unsuccessfully. *If a defective bucket is found then change the link of the preceding bucket to nil, and terminate unsuccessfully.* Otherwise return the object header that was found.

The search algorithm only checks the buckets that it touches during its normal course of

searching. In other words, when the search algorithm finds the object header in question, it terminates at that point, instead of continuing to check the remaining buckets towards the end of the chain. Any errors that are located further down the chain can be detected and corrected just as easily by the next operation that touches the final part of the chain, since the search algorithm does not modify the chain.

Next, the modified insertion algorithm is as follows (again, all changes and additions are italicized):

1. Perform the search operation on the object header.

2. If the search terminates successfully, finding an older version of the object header in some bucket B, then insert the updated version of the object header in B and terminate. Otherwise, hash the object header to the home bucket, X. Do one of the following:

    a. If bucket X is empty, or contains a deleted object header whose home bucket is bucket X, then simply insert the new object header into bucket X.

    b. If bucket X contains a valid object header whose home bucket is bucket X, then check for another bucket on chain X that contains a deleted object header. If one exists then insert the object header there. Otherwise, find another available bucket, Y, insert the object header in it, and add it to the end of chain X.

    c. If bucket X contains an object header whose home bucket is not bucket X, then bucket X must be part of another chain beginning with bucket Z. Thus, it is necessary to move the object header presently in bucket X to some other bucket. Starting with bucket Z, search down chain Z until either *a defective bucket is found,* a *non-defective* bucket containing a deleted object header is found, or until the end of the chain is reached. If a non-defective bucket, D, containing a deleted object header is found, then move the object header in bucket X to bucket D. *If a defective bucket is found then change the link of the preceding bucket to nil, and continue as if the end of the chain was reached.* If the end of the chain is reached, then move the object header in bucket X to a free bucket, F, and reroute chain Z through bucket F. Then, once the old object header has been removed from bucket X, insert the new object header there.

The insertion algorithm does not need to explicitly include a consistency check for chain

X because, as its first step, it executes the search algorithm (which checks for inconsistencies) on chain X. On the other hand, it does have to check through the buckets in chain Z. In fact, every bucket in chain Z must be checked, regardless of the relative position of bucket X in the chain (unlike the consistency check performed within the search algorithm). The reason for this is that chain Z is to be modified in such a way that may make a cycle invisible to the current cycle detection procedure, as was demonstrated on page 63.

This might lead one to believe that when performing an insertion, the check performed implicitly within the search algorithm on chain X is not sufficient because chain X may still contain a cycle when the insertion algorithm alters it. However, it is sufficient because if a bucket is found to contain the object header before the end of the chain is reached, no structural changes will be made to the hash table since the object header will only be reinserted in the same bucket. Thus, since nothing will be done to disturb any cycles or merges further down the chain, the next operation that executes on the chain will still be able to detect any inconsistencies. In addition, if the object header is *not* found to exist already in some bucket, then the search algorithm will have checked through the entire chain in the process of looking for the object header and will have corrected any inconsistencies that it found. For these reasons, it is not necessary for the insertion algorithm to include an explicit consistency check for chain X.

Furthermore, since the deletion algorithm does not make any structural changes to the hash table, it does not have to be modified at all. Thus, since it is comparable to the search algorithm in its requirements for error detection and correction and includes the search algorithm as its first step, the chain that contains the object header to be deleted will be implicitly checked and corrected.[11]

---

[11] The algorithm for finding a free bucket has not been described in detail because it searches through the disk pages in some optimum order with respect to disk access time and is fairly implementation specific. Even though it does not use chaining to guide its searches for buckets that can be freed up, whenever it actually removes a bucket from a chain, it must do a consistency check on the *entire* chain (as is done in the insertion algorithm) and break the chain if any defective buckets are detected. In this way, it will not modify a cyclic chain in such a way that would make the cycle transparent to the simple detection procedure.

Consider once again, the chain in Figure 4-11 on page 67. It is a merged chain consisting of chains 1 and 12, and also contains a cycle.If the insertion of oh27 had been done using the modified algorithms instead of the old ones described in Chapter 3, then the cycle would have been broken and the insertion would have proceeded properly. First, the search algorithm would have terminated unsuccessfully. Thus, oh81 would have been moved and its chain would have been rethreaded through the new bucket. In the process, each object header in chain 1 would have been checked in order to ensure that its home bucket was bucket 1. However, bucket 12 would have been found to contain oh72. Since oh72's home bucket is bucket 1, the link from bucket 7 to bucket 12 would have been changed to nil. Then, once the two chains that were merged had been separated and the cycle had been broken, as illustrated in Figure 4-15, oh81 would have been moved to another free bucket (since there were no buckets already on chain1 with deleted object headers). Finally, chain 1 would have been rerouted through the bucket containing oh81 and oh27 would have been inserted into bucket 7, forming a new, separate chain, as shown in Figure 4-16. Note, that even though the cycle was broken, chain 12 and chain 1 were still left merged at a second link between bucket 12 and bucket 3. It was not critical to correct this merge during the insertion of oh27, since the bad link would be broken the next time an object header was inserted in chain 12.

All other examples that were given in Section 4.1.1 would also have worked correctly if the modified algorithms were used. Since the changes made to the algorithms for searching and inserting object headers in the hash table ensure that the internal structure of the table is always correct or detectably incorrect, crashes cannot alter the behavior of the hash table algorithms. In other words, they cannot decrease the efficiency of the algorithms nor can they prevent them from terminating.

4.1.3 Obsolete, Lost and Duplicated Object Headers

There are two ways in which the object header table can be damaged, making it inconsistent with the current state of the object versions in VS. First, an object header can become obsolete if it is modified in the page buffer but a crash occurs before the page is

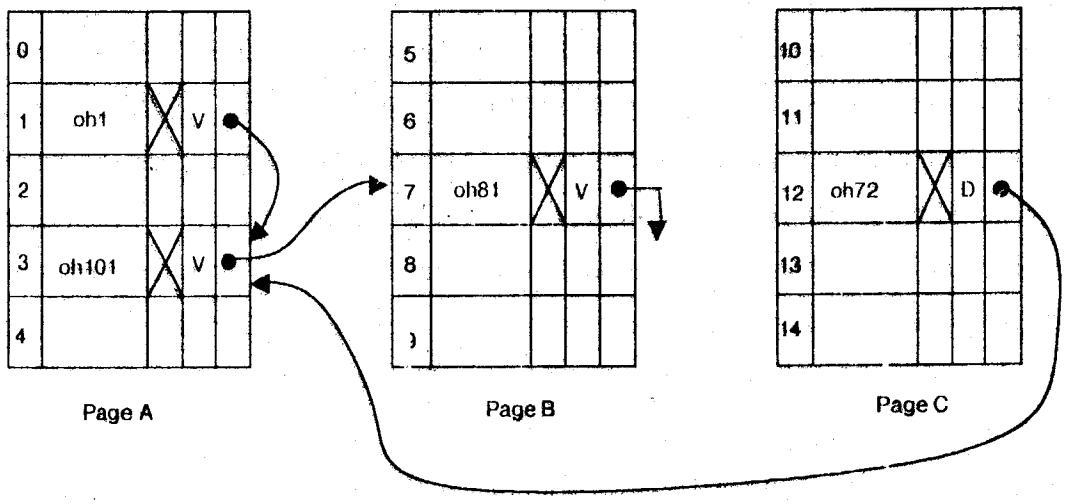


Figure 4-15: Separation of A Merged Chain

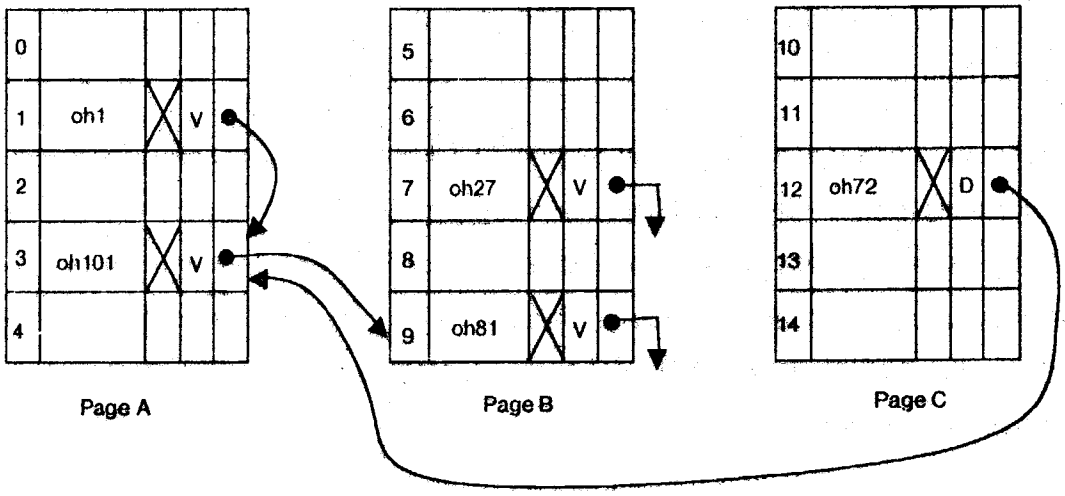


Figure 4-16: Pages A, B and C After Insertion of Oh81

written out to the disk in the modified state. Even though the object header appears to be valid, it contains out of date information about the object.

Second, an object header can get lost if a failure causes the OHS page on which it is located to go bad, or a failure occurs before all pages that have been modified by the insertion algorithm have been written from the buffer into OHS. For example, consider chain 1 in Figure 4-17 and suppose oh66 is to be inserted in the hash table.

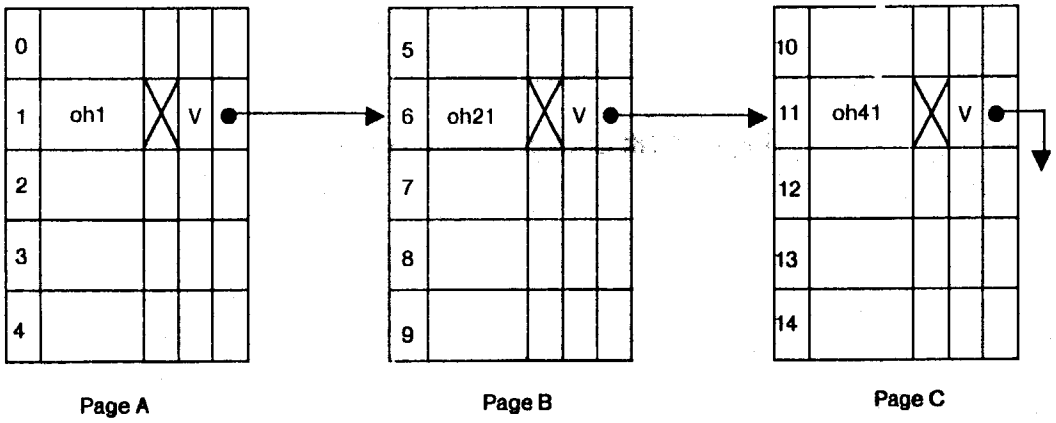


Figure 4-17: Pages A, B and C Before Oh 66 is Inserted

Before inserting oh66, oh21 has to be moved to another bucket and chain 1 has to be rerouted through that new bucket. The final state of the pages in the buffer, after the insertion is correctly executed, is illustrated in Figure 4-18. Now, suppose page B is written on the disk in its new state but the repository crashes before page A is written out. Both oh21 and oh41 become lost by virtue of the fact that they are no longer linked to the chain in which they belong, as shown in Figure 4-19. The normal search procedure will not find them because it will terminate after searching through buckets 1 and 6.

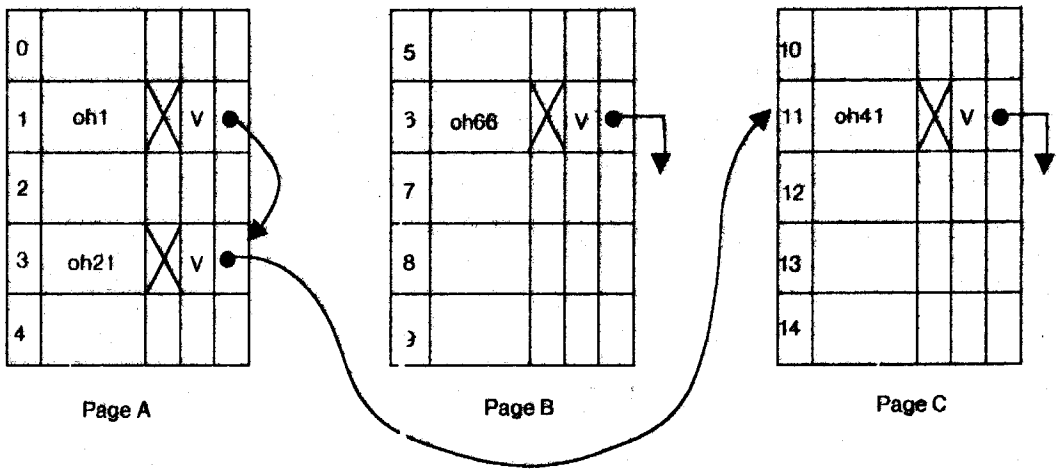


Figure 4-18: Correct Insertion of Oh66

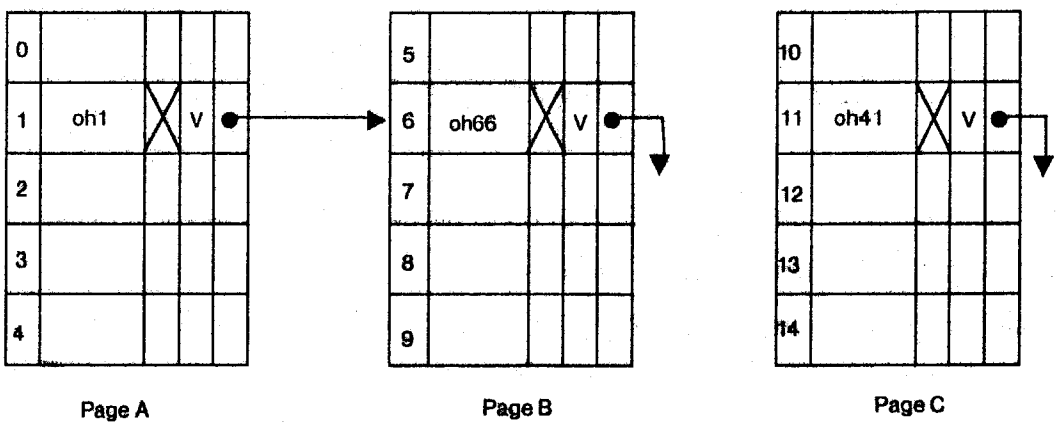


Figure 4-19: Pages A, B and C After Crash

4.1.4 Recovery of Lost and Obsolete Object Headers

In order to recover a lost or obsolete object header, the repository must restore the current version reference, the token reference, the commit record reference and the end pseudotime. First, the three references can be determined from the current version or token in VS, as follows. The repository searches sequentially backwards through VS from the VS write pointer until it finds a simple or root version image for the corresponding object. This version image will either be the current token, an aborted token or the current version. In order to determine which of three it is, the repository must check the state of the commit record that is referenced by the version image (assuming for now, that the commit record has already been recovered).

If the state of the commit record is UNKNOWN, then the version image is a token and the three references in the object header (current version reference, current token reference, commit record reference) should be set to the token's VS address, previous version reference and commit record reference, respectively. On the other hand, if the state of the commit record is ABORTED, then the version image is an aborted token and the object header's token and commit record references should both be set to nil. Furthermore, the previous version referenced by the aborted token is the object's current version so the current version reference of the object header should be set to point to that previous version image. Finally, if the state of the commit record is COMMITTED, then the version image is a version and there is no token. Thus, the three references in the object header should be set to the version image's VS address, nil, and nil, respectively (using the order above).

Now, the remaining value that the repository must restore in the object header is the end pseudotime of the current version or token. The repository simply sets this field to the pseudotime of recovery because that pseudotime is the earliest possible pseudotime at which it is *guaranteed* that no request has read the version or token. The exact, original end pseudotime may have been even earlier but cannot be easily determined by the repository. Thus, the pseudotime of recovery is satisfactory since it still ensures that all atomic actions are properly synchronized in their accesses to the object even though some atomic actions may be aborted unnecessarily due to the arbitrary extension of the end pseudotime. Section

4.3 will discuss how the repository determines the pseudotime of recovery.

Unfortunately, there are some complications to the recovery of lost and obsolete object headers. First of all, the repository cannot discriminate between an obsolete and current object header, using only the information in the object header. Second of all, the repository does not have any bound on its search through VS, when recovering a lost or obsolete object header. Since VS is always increasing in size, it is not acceptable for the repository to do an unbounded search every time it has to recover an object header. Thus, we need a means for *detection* of obsolete object headers and an efficient means for *correction* of both obsolete and lost object headers. For these reasons, *recovery epochs* and *OHS checkpoint epochs* have been developed.

4.1.5 Recovery Epochs

A recovery epoch is the time period between two repository crashes. Each recovery epoch is distinguishable from the others by its *recovery epoch number*, or REN, which is a monotonically increasing number. Whenever the repository crashes and restarts, it increases its REN, which it maintains as part of its global state. Also upon restarting, the repository marks the beginning of the new recovery epoch in VS by writing (in VS) a boundary version image, called a *recovery epoch mark*, or REM, which contains the new REN. This REM enables the repository to determine in which recovery epoch any version image was created.

Now, in order to determine whether an object header is current or obsolete, the repository must check that the object header contains a reference to the most current simple or root version image of the object in VS. If the object header does not contain a reference to the most current version image of the object, then the repository must update the object header. However, the repository only has to check each object header once per recovery epoch, since it marks the object header with its current REN after the first check. Thus, whenever an object header is accessed, its REN is compared to the repository's current REN. If the two REN's are the same then the object header is current. Otherwise, the object header is either obsolete or is still current as of the new recovery epoch but has not

been accessed since the last time the repository crashed. Therefore, if the REN of the object header is not the same as that of the repository, then the object header must be *certified* to be current.

The recovery manager, which will be discussed in Section 4.3, is responsible for certifying the object headers. In order to certify an object header, the recovery manager scans sequentially backwards through VS, searching for a more current simple or root version image of the object than the version image referenced in the object header. If it finds one, then it updates the object header's references and end pseudotime and marks the object header to be current by setting the object header's REN to that of the repository. If the recovery manager does not find one, then it just sets the object header's REN to that of the repository, since the object header is still current.

In order to certify a potentially obsolete object header, the recovery manager only has to search through the portion of VS that is bounded by the REM of the current recovery epoch[12] and the REM of the recovery epoch that corresponds to the object header's REN. The recovery manager does not have to search through the current recovery epoch in VS because if the object header had been accessed in this epoch its REN would be current. Furthermore, the recovery manager does not have to search past the REM that corresponds to the recovery epoch of the object header's REN since that REN indicates that the object header was last certified to be current in that recovery epoch. Thus, if the recovery manager does not find a version for that object by the time it reaches this REM in VS, then the object header is still current, and the recovery manager only has to update the object header's REN.

The recovery manager's search through VS can be further minimized if recovery epochs are artifically created whenever *all* OHS pages that have been modified in the buffer have been written out to the disk (i.e., when the repository becomes idle), and if each REM is marked as either a *crash* or *non-crash* REM. Using this scheme, the recovery manager

---

[12]The current recovery epoch is the new recovery epoch that began when the repository restarted after the most recent crash.

would only have to scan through the non-crash recovery epoch immediately preceding the crash recovery epoch. For example, suppose that the repository crashes during recovery epoch 8 and upon restarting, writes a crash REM for recovery epoch 9 into VS, as shown in Figure 4-20. If an object header with an REN equal to 5 is accessed after the crash, then the recovery manager only has to scan through recovery epoch 8 for a more current version image because all OHS pages that were modified during recovery epochs 5, 6, and 7 are known to have been written out by virtue of the fact that they all precede another non-crash recovery epoch.

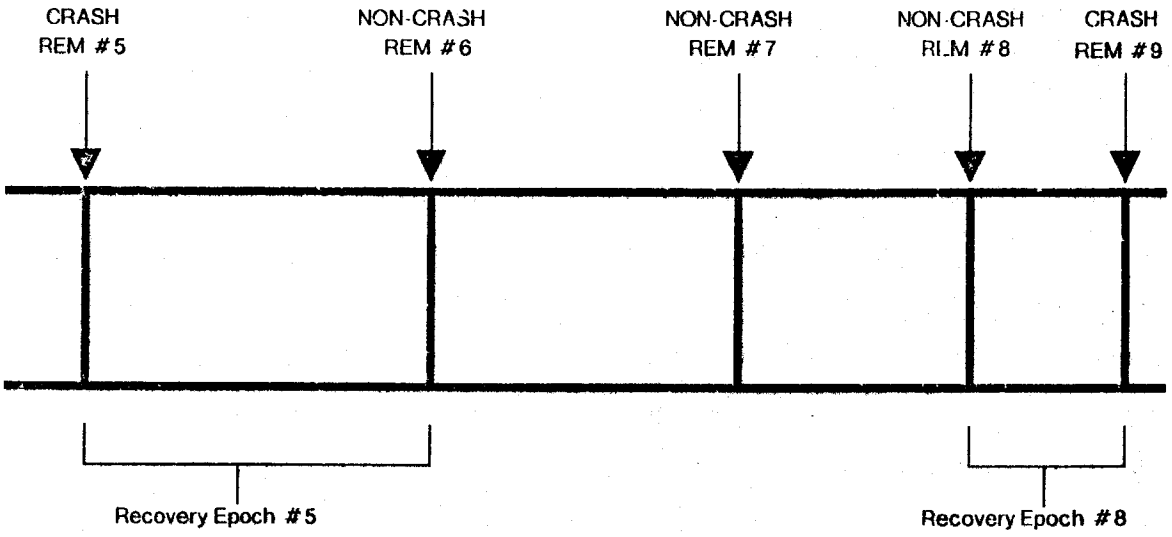


Figure 4-20: Recovery Epochs In VS

Thus, the benefits of recovery epochs are twofold. Each object header only has to be checked for obsolescence once per recovery epoch and when it does have to be checked, the search through VS for the current version image is bounded.

4.1.6 OHS Checkpoint Epochs

Even though recovery epochs exist, there is still a problem in bounding the recovery manager's search for the current version image of a lost object header, since there is no

object header to provide an REN. For example, if the OID of a supposedly lost object header had never been assigned to any object because of a crash, then there would not be any version images in VS with that OID and the recovery manager would have to search through all of VS before it could finally figure this out. Similarly, if the object corresponding to the lost object header is very old, then in order to find the current version image, the recovery manager would have to search through a large portion of VS. In order to prevent these unbounded searches, a table that checkpoints the object header information for every object that is current, is periodically created in VS, thereby enabling the recovery manager to bound its search through VS with the location of the most current completed table. This table is called a *checkpoint table* and the period of time over which it is created is called a *checkpoint epoch.*

Each entry in a checkpoint table consists of the object's OID as well as a reference to the version that is current at the time the entry is created. Since the construction of a checkpoint table may consume a large amount of time, it is not acceptable for the repository to temporarily discontinue service in order to take a snapshot of the state of all object headers at one specific point in time. Instead, the checkpoint table is created in the background by a separate process, called the *checkpoint manager* while the repository accepts and services external requests. Thus, a checkpoint table does not necessarily capture the current state of every object header at one particular point in time but instead, it captures *some* state that was current for each object header at *some* time during the checkpoint epoch in which the table was created. Further, since the checkpoint table is created in VS while versions are also being created, its entries may be interleaved with the versions. Thus, all of the checkpoint entries are linked together in order to make it possible to search through the version images of the checkpoint table exclusive of the rest of VS. Finally, before the checkpoint manager starts to create a new table, it writes a *checkpoint epoch mark*, or CEM, in VS in order to mark the beginning of the new checkpoint epoch.

The checkpoint manager has to be sure to include an entry in the checkpoint table for every object that existed during that checkpoint epoch. However, the checkpoint manager would not necessarily do so if it simply created a checkpoint entry for every object header in

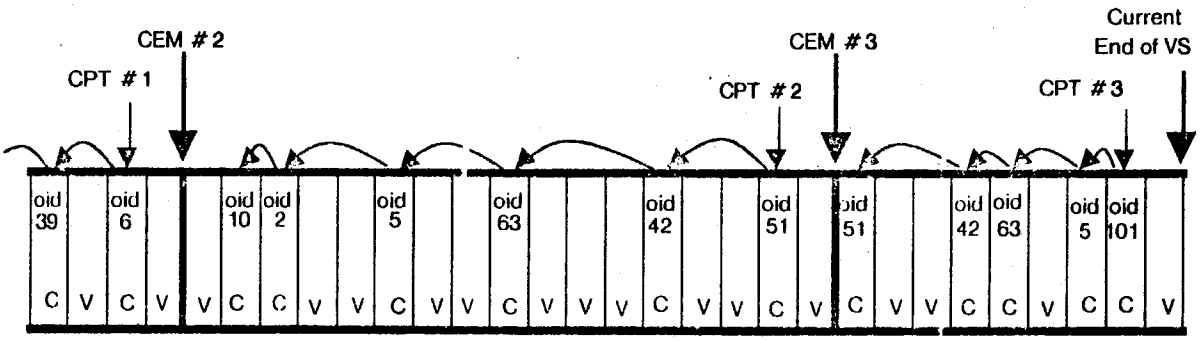
OHS since some object headers may have been lost. On the other hand, it would do so if it searched through VS for all of the current version images and created a checkpoint entry for each one it found. However, this would be at least as bad as searching through VS for every lost object header, if not worse. Since the original reason for creating a checkpoint table was to minimize and bound the recovery manager's search through VS, the checkpoint manager should not have to make an unbounded search in order to create the table. Therefore, it was necessary to come up with some other scheme that would account for every object that existed during a checkpoint epoch.

The checkpoint manager creates the checkpoint table, as follows. When each object is first created in the repository, the checkpoint manager creates a checkpoint entry for it in the current checkpoint table. Then, the checkpoint manager accounts for the remaining objects existing in the checkpoint epoch by updating each entry that exists in the checkpoint table of the previous checkpoint epoch, and placing it in the new checkpoint table.

In order to update an old checkpoint table entry, the checkpoint manager examines the corresponding object header and extracts the reference to the current version or token. However, if the object header is lost or obsolete then the checkpoint manager must wait until the recovery manager certifies the object header before updating the checkpoint entry. Also, if the object header indicates that the object was deleted in the previous checkpoint epoch, then the checkpoint manager does not write any updated entry for it in the new checkpoint table. Thus, it can be seen that this method of creating successive checkpoint tables from previous ones is guaranteed to include entries for all objects that ever existed in each checkpoint epoch, without having to scan through all of the version images in VS.

When searching for the current version image of an object whose header appears to be lost, the recovery manager should either find an actual version or a checkpoint table entry containing a reference to the current version, by the time it reaches the CEM of the last completed checkpoint epoch in VS (which will be referred to as the limiting CEM). Otherwise, the object has been deleted in some previous checkpoint epoch or it never existed. For example, consider the checkpoint epochs in VS that are illustrated in Figure 4-21. Since the table for checkpoint epoch #3 is still being created, checkpoint table #2 is

the last completed checkpoint epoch. This means that CEM #2 is the limiting CEM and thus, the recovery manager would only have to scan through to CEM #2 before it could conclude that an object never existed or was deleted. Thus, CEM's provide the lower limit for the recovery manager's search for lost object headers.



C = checkpoint table entry
V = any other type of version image
CEM #X = beginning boundary of checkpoint epoch #X
CPT #X = checkpoint table for checkpoint epoch #X

Figure 4-21: Checkpoint Tables In VS

In order for CEM's to be valid limits for the recovery manager's searches through VS, the repository's processes must never confirm the creation of an object (to an external node) until the checkpoint manager confirms that a checkpoint entry has been created in VS, since if they did, then it would be possible for a crash to occur after a confirmation was sent out but before the entry was made for it in the checkpoint table. In other words, it would be possible for an object to exist without having a corresponding entry in the checkpoint table and the recovery manager might incorrectly conclude that such an object never existed, if the corresponding object header ever got lost.

For example, assume that the creation of object A is confirmed to an external node before the checkpoint manager confirms the creation of the corresponding checkpoint entry, and that the repository then crashes before the entry is created. Since each

subsequent table is created from the previous table, an entry for object A will never be created in any of the checkpoint tables. Now, if the object header for A gets lost at some time when the current version of A is located further back in VS than the limiting CEM (see Figure 4-22), and the repository receives a request to read A, then the recovery manager, in attempting to find its current version in VS, would incorrectly conclude that the object was previously deleted or never existed, since it would not find a version or checkpoint entry by the time it reached the limiting CEM. However, if the process that created object A had not sent out the confirmation in the first place, then the object still would have been nonexistent and no external node would have sent any requests for it.

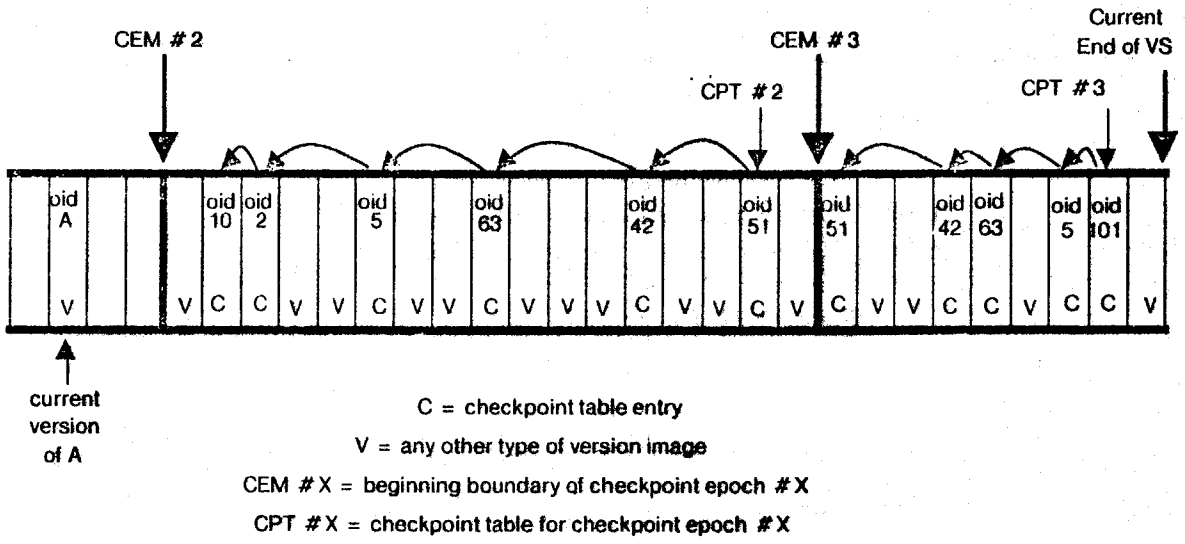


Figure 4-22: No Checkpoint Entry for Object A

Finally, there is one decision that still has to be made concerning checkpoint epochs, that is how often should the checkpoint manager start a new checkpoint epoch? The only constraint is that a new epoch cannot be started until the checkpoint manager has made updated entries in the current table for all of the entries in the previous checkpoint table. As long as this requirement is met then the checkpointing mechanism will work correctly. The discussion of how this decision should be made is deferred until Chapter 5, which

analyzes the costs of the necessary tradeoffs.

4.2 Recovery of Commit Records

Now that the recovery of objects has been described, it is necessary to explain how the commit records are recovered. Recall that the versions of the commit records (as well as those of the objects) are maintained in VS, which is stable storage. Thus, VS in itself also contains the current state of the commit records. However, the repository accesses these versions of a commit record through the commit record table, which is only kept in volatile storage. Thus, when the repository crashes, the commit record table is completely lost.

Upon restarting, the repository creates an empty table and adds entries as new commit records are created. Also, when the repository restarts after a crash, it implicitly aborts all commit records that were in the UNKNOWN state at the time of the crash, since there is a good probability that the broker that created the commit record would have aborted it anyway, due to the crash. However, this abortion is not done explicitly, since the commit record table no longer contains entries for any of the commit records that were created before the crash. Instead, the abortions are done as follows.

As the recovery manager scans sequentially through VS in order to recover object headers, it creates entries (unless they already exist) in the commit record table for any version images that it encounters that contain the final state of a commit record. However, it only creates these entries if the actual state is COMMITTED.[13] Thus, when the recovery manager is actually recovering an object header, if the corresponding token's commit record is not found in the commit record table then that commit record has been aborted. Either the recovery manager had found the final state version before it reached the token in VS but did not create an entry for it in the commit record table since its state was ABORTED, or else there was no final state version in VS and thus, the commit record was aborted by

[13] Asynchronously, some other process will eventually delete these entries from the commit record table, after rechecking that the object headers of all tokens in the linked list have been updated to reflect the token's (commit record's) final state.

definition.

Furthermore, the broker that created a commit record that was automatically aborted is eventually informed of this automatic abortion when it attempts to retransmit any unconfirmed create-token requests, or tries to set the final state of the commit record. Upon being informed, the broker retries the entire atomic action. Thus, the repository's recovery of commit records supports Swallow's atomic action protocol.

4.3 Recovery Manager

This section describes how the recovery manager coordinates the repository's recovery activities and interfaces with the other repository processes when they access object headers that have to be recovered. In a nutshell, the recovery manager restores the repository to a state in which it can resume servicing requests from other Swallow nodes after a crash and then runs in the background, during the repository's normal course of activities, certifying the object headers and temporarily creating entries in the commit record table that facilitate the recovery of these object headers.

Thus, when the repository restarts after a crash, it does not start accepting messages until the recovery manager signals that the global state data has been properly updated in State Storage and encached in volatile storage. However, once this signal is received, the repository resumes its communications with the other Swallow nodes.

The only values in State Storage that the recovery manager has to update are those of the VS write pointer, the last uid assigned to an object or commit record, the latest pseudotime specified by any request, and the repository's REN. In order to simplify the description of the recovery of the values of the VS write pointer and the last uid assigned, several terms are defined as follows:

WP = value in State Storage of the VS write pointer

X = the number of pages that must be written in VS before WP is updated in State Storage

LUA = value in State Storage of the last uid assigned to an object or commit record

Y = the total number of uid's that must be assigned to objects and commit records before LUA is updated in State Storage

Both WP and LUA are periodically written into State Storage but the active copies are updated in volatile storage. The values X and Y above control the frequency and thus the cost of State Storage updates. Conversely, they also control the cost of recovery.

In order to restore the VS write pointer, the recovery manager must search sequentially through the region in VS, bounded by the two pages, WP and WP+X, until it finds the last VS page that has been written. Furthermore, in order to restore the current value of last uid assigned to an object or commit record, the recovery manager simply assumes that Y uid's were actually assigned before the crash, and increases LUA by Y. In this way the repository is still guaranteed to assign *unique* id's to the objects and commit records even though some uid's will never be assigned. (Since the uid is a 64 bit number, it is not critical if some uid's are wasted.)

Thus, X and Y are tuning parameters. A large X value increases recovery time and a large Y value increases the waste of uid's upon failure. Balancing these costs against the cost of State Storage updates should be simple.

Next, the recovery manager must restore the latest pseudotime specified by any request since this pseudotime is used as the pseudotime of recovery. Although the working copy of this value is kept in volatile storage, it is also stabilized by recording, on each VS page, the value of the latest end pseudotime of all versions in VS up to and including that page. Thus, upon restarting, the recovery manager simply accesses this value from the last VS page written into VS.

The remaining value of the global state that the recovery manager must update is the repository's REN, since the REN found in State Storage is obsolete due to the fact that the repository just crashed. Thus, the recovery manager increments the value of the REN found in State Storage and writes an REM for the new recovery epoch into VS. Furthermore, a volatile copy of the current REN is maintained in primary memory to speed up the process of checking object headers.

There is one final task that the recovery manager must perform before signalling that it is safe for the repository to accept external messages. It must restore the checkpoint manager, since the checkpoint manager must continue creating the current checkpoint table from where it left off when the repository crashed. To speed recovery, it is arranged that every VS page contains a pointer to the most current checkpoint entry written into VS. Thus, in order to restore the checkpoint manager, the recovery manager obtains from the last page written into VS, the location of the last checkpoint entry that was written into VS and passes it on to the checkpoint manager. Then, the checkpoint manager can actually access that checkpoint entry, find the most current checkpoint entry (in the current checkpoint table) that is also in the previous checkpoint table and resume updating the entries in the new table, starting with that checkpoint entry in the previous table.

For example, assume that the repository is recovering after a crash and that the state of VS is as depicted in Figure 4-21 on page 82. In this case, the recovery manager would pass the VS address of the checkpoint entry for object 101 to the checkpoint manager since it contains the most current checkpoint entry that was written into VS. Then, the checkpoint manager would determine that the checkpoint entry for object 5 is the most current checkpoint entry written into VS that is also found in the table created in checkpoint epoch #2, and thus, would continue creating the table for checkpoint epoch #3, by starting with the entry for object 2 found in the table for checkpoint epoch #2.

Once the recovery manager completes all tasks, described thus far, the repository begins to accept and fulfill external requests, even though some of the repository's data may still be incorrect. Thus, when another repository process accesses an object header that is lost or contains an old REN, it must wait until the recovery manager certifies the object header.

Once the recovery manager completes the certification or concludes that the object header corresponds to a deleted or non-existent object, it signals the waiting process. That process then reaccesses the object header and simply continues with its regular tasks, if the object header exists. However, if the object header is still lost, then the process (like the recovery manager) concludes that the object header corresponds to a deleted or non-existent object and takes the appropriate alternate action.

In order to avoid repetitious scanning through VS, the recovery manager certifies the object headers for *all* version images that it accesses as it searches sequentially backwards through VS. In addition, during this sweep through VS, the recovery manager temporarily creates entries for *all committed* commit records that it encounters and removes the entries when the scan passes the initial commit record versions, since earlier versions will not access the commit record. Then, when the recovery manager is recovering an object header and is trying to determine whether the current version image is a token or a version, it knows that the version image is a token if there is an entry in the commit record table for the corresponding commit record, and conversely, knows that the version image is an aborted token if there is no entry in the commit record table.

However, the recovery manager does not perform its scan through VS continuously in the background until it finishes. Instead, it scans through VS, certifying all corresponding object headers until there are no more processes waiting for object headers to be certified. Then it halts temporarily, remembering where it left off in VS and resumes either when the repository becomes idle (has no pending requests), or when some process needs another object header that has not yet been certified, in order to fulfill a request.

Thus, while the repository has pending requests, the recovery manager only has to search through the non-crash recovery epoch that precedes the most recent crash recovery epoch, providing that there are no lost object headers. Only in the rare cases where an object header is lost would the recovery manager have to search through VS up to the limiting CEM while the repository has pending requests. However, since the recovery manager continues to certify object header during the repository's idle periods, lost object headers may be recovered before they are required in order to satisfy a request, and thus, their

recovery will not affect the repository's response time for fulfilling requests.

Assuming, for now, that disk failures do not occur in OHS while the recovery manager is in the midst of certifying the object headers, the object header table will be completely valid after the recovery manager has made one scan through all of of VS up to the limiting CEM. Thus, it can signal all processes that are still waiting for lost object headers to be certified after it completes this scan, and these processes will correctly conclude that the object headers correspond to objects that were deleted.

However, there is a problem with this reasoning due to the fact that disk failures are not controlled events and can occur any time. Usually, when the repository detects a disk failure (bad page) in OHS, it crashes itself and restarts all of its recovery mechanisms since OHS may no longer be consistent with VS. But while the recovery manager is certifying all of the object headers, the repository cannot determine whether a bad page is the result of a disk failure from which the repository is presently recovering or whether a subsequent disk failure occurred. Therefore, the repository does not crash itself if it encounters a bad page in OHS if the recovery manager is still certifying the object headers.

This means that the recovery manager can no longer simply signal any processes that are still waiting for lost object headers, after it completes its initial search through VS, since any of these processes could be waiting for an object header that is on a disk page that was destroyed by a disk failure that occurred after the recovery manager certified that object header. Thus, if (and only if) there are still processes waiting for lost object headers to be certified after the recovery manager makes its initial scan through VS, then the recovery manager must recheck the object headers for all of the current objects. That is, it must check the object headers that correspond to the checkpoint entries up to the limiting CEM. Then, if all of the object headers are still valid, the recovery manager can signal any processes that are still waiting. However, if the recovery manager encounters a bad disk page in OHS during this second scan, then the repository will crash itself and restart its recovery mechanisms so that it can, once again, restore consistency between OHS and VS. Furthermore, the repository will crash itself if *any* repository process encounters a bad OHS page *after* the recovery manager makes its initial scan through VS.

Thus, the portion of VS through which the recovery manager may have to scan *while the repository is servicing external requests*, depends upon the extent of the damage that is done to OHS. First, if no object headers are lost then the recovery manager only has to search through the non-crash recovery epoch that precedes the most recent crash recovery epoch. Second, if some object headers are lost, then the recovery manager has to scan through to the limiting CEM. Finally, if some process tries to access an object header on an OHS page that has gone bad since the recovery manager recovered that object header, or tries to access a non-existent or previously deleted object header, then the recovery manager not only has to search through to the limiting CEM, but also must reaccess all the checkpoint entries up to that limiting CEM (in order to recheck their corresponding object headers).

4.4 Justification for Lack of Recovery of Pending Messages

Since all data describing the pending messages is kept in volatile storage, when the repository restarts fter a crash, all this data is lost and the repository is left with no recall of the prior state of these messages. However, the repository does not have to remember the prior state of pending messages since it does not continue to process these messages from where it left off at the time of the crash. Instead, upon restarting, it accepts new messages and starts from scratch.

Now that all of the repository's recovery mechanisms have been described in detail, it is possible to explain why the repository does not have to explicitly recover its pending messages after a crash. Basically, there are three reasons. First, since the repository satisfies all requests atomically, no data will remain partially modified. The data will either be completely modified or not modified at all. Second, since the protocols include provisions for any communications errors that might occur, both the sender and receiver of the message know exactly how to react when any of these errors occur. Finally, since all repository requests are repeatable, as demonstrated in the table in Figure 4-23, retransmissions do not cause the same modifications to be done twice to the same data.

The following example, in which the consequences of not recovering a multiple packet

1. **Create-Object**: In order to decide whether or not a create-object request is a retransmission, the repository would have to search through VS for a version that contains the same pseudotime and commit record id as those named in the request. However, this is totally unnecessary, since the original request was unconfirmed the requestor does not have the oid and cannot access the object that the repository originally created. Therefore, for all intents and purposes the object still does not exist, so the repository can create a new object when it receives a retransmitted create-object request in the same way as if the request was not a retransmission.

2. **Delete-Object**: If the object is already deleted when the request is received then repository just confirms the deletion. Otherwise, the deletion is performed.

3. **Read-Version**: Does not modify data, so retransmission is confirmed in exactly same way as original request.

4. **Create-token**: When the repository receives the retransmission and tries to create the token it will find that a token already exists in the object history. It checks whether or not the request is a retransmission by checking the pseudotime and commit record id of the token. If they are the same as the pseudotime and commit record id named in the request then it knows that this request is a retransmission and has already been satisfied. The repository simply confirms the creation of the token.

5. **Test-Commit-Record**: Same as Read-Version

6. **Abort-Commit-Record**: Once state of commit record is decided it is never changed so repository will simply respond with the final state of the commit record.

7. **Commit-CommitRecord**: Same as Abort-ComRec

8. **Add-Reference**: Repository will not add a representative version to a commit record's reference list if that version is already on the list. Repository will simply respond with confirmation that reference has been added.

9. **State-Is**: If the repository has not already encached the final state in the commit record representative then it does so. Then it returns a delete reference response (even if the state had already been encached).

Figure 4-23: Handling of Retransmitted Requests

create-token request are described, should demonstrate that these reasons are valid. Since a create-token request may be left in one of four inconsistent states after the repository crashes, the example will consist of four explanations, one for each possible state.

State 1: The repository only received the initial packet of the message but had not yet begun to process it. Furthermore, the repository did not send any response to the broker.

Since no data was modified, there are no inconsistencies in the repository's data. Furthermore, since a confirmation was never sent to the broker, the SMP module at the broker's node will eventually time out and abort the message, at which point the broker will either abort the atomic action (send an abort-commit-record request) or retransmit the request. Subsequently, the repository will either start from scratch if the broker retransmits the request, or will abort the commit record as usual, if the broker sends an abort-commit-record request.

State 2: The repository received some or all of the packets but did not write all of the VS pages containing the version. Furthermore, the repository did not make the necessary modifications to the object header table nor did it send any response to the broker.

In this case, the token still does not exist since the root version image, which is always located on the most current VS page containing the token, was never written. In addition, the token is not linked into the commit record's list of tokens since the root version image is the only version image of the token that contains the link. Furthermore, since the object header table was not modified, the object header still points to the current version. During recovery, the recovery manager will not change the object header to point to the partially written token because it will not find a root version image and ignores the fragment version images. Finally, since the confirmation was not sent to the broker, the broker will either abort the atomic action or retransmit the request and the repository will react in the same way as was described for State 1.

State 3: The repository received all packets and wrote all VS pages containing the token. Thus, by definition, it also added the token to the commit record's list. However, it made some or no modifications to the object header table and did not send a response to the

broker.

In this case, the recovery manager will eventually update the object header to point to the newly created token and the hash table algorithms will restore consistency to the object header table. Furthermore, since no confirmation was sent to the broker, the broker will either abort the atomic action or retransmit the create-token request. If the broker sends an abort-commit-record request, then the repository aborts the commit record (if it has not already been aborted by the recovery manager) and confirms the request. On the other hand, if the broker retransmits the create-token request, then the following sequence of events occurs. First, the repository process that is handling the request accesses the appropriate object header. If the recovery manager has not yet recovered the object header, then the process must wait until the recovery manager signals that the object header has been certified. Then, when the process reaccesses the object header it creates a token since the recovery manager deleted the existing one[14] and attempts to add the token to the appropriate commit record's list of versions. However, in attempting to add the token to the commit record's list, the process discovers that the commit record has been aborted. Thus, the process deletes the token and sends a rejection reponse to the broker, specifying that the commit record has been aborted. Subsequently, the broker will retry the entire atomic action.

State 4: The repository received all packets and made all of the necessary modifications, but did not send a confirmation to the broker.

The repository handles this state in the same way as it handles State 3.

Thus, it can be seen from this example that all inconsistencies in the repository's data caused by partially processed create-token requests are eliminated by the repository's recovery mechanisms. Furthermore, the broker is not left hanging when the repository fails to respond, since the SMp, request/response and atomic action protocols provide alternative

---

[14]When the recovery manager recovers the object header the commit record will have been aborted. Thus, the recovery manager deletes the token that was created when the original create-token request was received, by changing the object header's token referenc to nil.

modes of behavior. In fact, for all types of messages that may be sent to the repository, the combination of the repository's internal recovery mechanisms and the Swallow protocols ensure that the global consistency of all clients' objects is restored.

4.5 Summary

Thus, the recovery mechanisms used to restore order within the repository were presented in this chapter. First it was shown how the structure of the object header table is recovered implicitly, using a special set of hash table algorithms, instead of by performing an exhaustive consistency check on the entire table structure right after a crash. Next, it was shown how the object headers themselves are recovered from the current versions in VS, using the recovery and checkpoint epoch mechanisms in order to determine the need for recovery and to bound the linear searches through VS. Then, it was shown how commit records are implicitly aborted if their state was not finalized before the repository crashed, and how committed commit records are temporarily entered in the new commit record table in order to speed recovery of the object headers. Finally, it was shown how the recovery manager restores the repository's global state as well as how the recovery manager coordinates all of the recovery activities so that it only has to perform a single scan through VS.

Chapter Five

Evaluation of Recovery Mechanisms

The effects of the recovery mechanisms on the performance of the repository are evaluated in this chapter. However, since the repository has not yet been implemented there are no real statistics on how long it takes the repository to satisfy the various types of requests. Still, it is possible to estimate these time costs in terms of the number of underlying disk accesses that must be done in order to do recovery and fulfill requests. This is a useful method of analysis since these disk accesses are likely to be the most time consuming tasks that the repository performs.

First, Sections 5.1 and 5.2, derive equations that calculate the total number of disk accesses that the recovery and checkpoint managers, respectively, require per recovery epoch. Next, Section 5.3, calculates the average cost of of these recovery mechanisms per request, for a typical example. From this calculation it is possible to gain some insight into how much of the repository's response time can be attributed to the recovery mechanisms and how sensitive these response time costs of recovery are to the varying characteristics of the requests and data sent to the repository. Finally, in order to put these calculations into perspective, Section 5.4 compares the cost of the recovery mechanisms presented in this thesis (for the repository) with an alternate set of recovery mechanisms that could have been used, which are based upon OHS being reusable stable storage.

5.1 Cost of Recovery Manager

The cost of the recovery manager includes the cost of updating State Storage and encaching it in Volatile Storage as well as the cost of certifying all of the object headers. Since the significant cost is that of certifying the object headers, this cost will be analyzed in detail, but first, a brief description of the other costs is given, as follows.

The only noticable cost of recovering State Storage (with respect to disk accesses) is that of restoring the VS write pointer, since the recovery manager has to search through some number of pages in VS in order to find it. This number depends upon how frequently the value of the VS write pointer is updated in State Storage; the more frequently the value of the VS pointer is updated in State Storage, the fewer the number of VS pages through which the recovery manager must search after a crash will be. However, State Storage updates are fairly costly (in terms of disk accesses) and should not be done too often while the repository has pending requests. Thus, a tradeoff must be made. In the initial implemention of the repository, the tradeoff will be made arbitrarily and then, once actual costs can be measured, the parameter that specifies the frequency of updating the VS write pointer in State Storage will be fine tuned for the optimum tradeoff.

The remaining costs of restoring State Storage depend on its size and what percentage of it must be encached in volatile storage. However, since State Storage will be fairly small (less than one page), these costs should be insignificant compared to the cost of recovering the write pointer.

In order to derive an equation for the total cost of certifying all object headers in OHS per crash, it is necessary to define the following variables:

C_{vr} = the cost of reading a VS page

C_{vw} = the cost of writing a VS page

C_{or} = the cost of reading an OHS page

C_{ow} = the cost of writing an OHS page

X = the number of OHS pages that have to be read in order to find a particular object header (using the hash table search algorithm)

P = average number of version images per VS page

L = probability that any object header will get lost during a checkpoint epoch

M_{re} = the REM (beginning mark in VS) of the non-crash recovery epoch that precedes the crash recovery epoch

M_{ce} = the limiting CEM (i.e. the beginning of the last terminated checkpoint epoch)

D = the number of pages in the portion of VS between M_{re} and M_{ce}

N = the number of VS pages in the non-crash recovery epoch that precedes the crash recovery epoch

I_N = the number of version images per N

V_N = the number of version images that are simple versions or roots of structured versions for objects per N

O_N = the number of distinct objects for which there are version images contained within N

E_N = the number of checkpoint entries per N

I_D = the number of version images per D
($I_N \ll I_D$)

V_D = the number of version images that are simple versions or roots of structured versions for objects per D
($V_N \ll V_D$)

O_D = the number of distinct objects for which there are version images contained within D
($O_N \ll O_D$)

E_D = the number of checkpoint entries per D
($E_N \ll E_D$)

ΔE_D = the number of new checkpoint entries that have been created between repository restart time and the time when the recovery manager finishes its initial scan through VS (up to the limiting CEM)

Using the above definitions, the *basic* total cost, C_{rm}, of the recovery manager per crash *assuming that no object headers are lost* can be specified:

$$C_{rmb} = C_{vr} I_N / P + X C_{or} (V_N + E_N) + C_{ow} O_N$$

The terms of the equation can be explained as follows. The first term in the equation reflects the cost of reading and examining every version image within N. Since the recovery manager scans sequentially through VS, it examines all of the version images on a single page while that page is in the buffer. Thus, the cost of examining the version images is reduced by a P factor due to the fact that the recovery manager does not make a disk access every time it examines a version image.

The second term represents the cost of reading the object headers corresponding to every version image that is a simple or root version image of an object, or a checkpoint entry, in order to check that the object headers are current. The cost of reading an OHS page is multiplied by X because in order to find a particular object header, the search algorithm must be executed on the object header table, which might involve reading more than one OHS page if the object header being accessed is on a chain that crosses page boundaries or

was damaged. However, very few (if any) chains in the object header table will have these properties since all of the buckets on a single chain are almost always located on the same page. Thus, the value of X is so close to 1 that for all analyses in this chapter it will be assumed to be 1.

The final term represents the cost of the OHS writes that must be done in order to update every object header. This term accounts for every object header being written once since it is assumed that the recovery manager reaches M_{re} before the repository crashes again.[15] Thus, for any recovery epochs for which this assumption is not true, this term will have to be adjusted. Furthermore, the cost of the OHS write in this term is not multiplied by a factor similar to X since the recovery manager retains the location of the object header when it first executes the search algorithm and can simply rewrite the object header in place without having to perform the insertion algorithm.

Since object headers sometimes do get lost, C_{rmb} is not the *average* total cost of the recovery manager per crash. In order to calculate this cost it is necessary to add to C_{rmb}, some percentage of the cost of scanning between M_{re} and M_{ce}. This percentage, L, represents the probability that a crash will cause object headers to get lost. Thus, the average total cost, C_{rmt}, of the recovery manager per crash is:

$$C_{rmt} = C_{rmb} + L\{C_{vr} I_D/P + XC_{or}(V_D + E_D) + C_{ow} O_D + [(C_{vr} + XC_{or})(E_D + \Delta E_D)]^{*}[16]\}$$

In the factor multiplied by L, all terms except for the starred term are costs that are comparable to the costs in C_{rmb}. The only difference is that the scan through VS is done

---

[15] An object header is never written more than once, even if there is more than one version for the object contained within the recovery epoch in VS, because once an object header has been certified it contains a current REN. The recovery manager does not rewrite any object headers that contain current REN's.

[16] Throughout the remainder of this analysis, the reader can assume that any term that is marked with an asterisk, * , is included in the cost only in the worst case. A very low probability event has to occur for the term to be relevant.

through the region bounded by M_{re} and M_{ce} instead of through the non-crash recovery epoch that precedes the lates crash recovery epoch. Furthermore, the starred term represents the cost of rechecking (second scan through VS) all of the object headers for all of the current objects. Recall that the recovery manager only does this if, after it initially checks and certifies all of the object headers, there still remain processes waiting for lost object headers to be recovered (see page 89). Only if one or more OHS disk pages decayed or if some external request erroneously specified an OID for a deleted or non-existent object will there be processes waiting after the initial scan. Thus, since both of these events occur very rarely, this starred term will not usually be calculated into the cost.

Thus, C_{rmt} is not only the average total cost of the recovery manager per crash but is also the average response time cost of the recovery manager per crash. In other words, it represents the cost of the work that the recovery manager must do in the background while the repository is satisfying external requests. However, keep in mind that C_{rmt} is the worst case average cost, since the repository may have idle periods in which the recovery manager can do some of the object header certification. In Section 5.3 it will be shown how C_{rmt} affects the average response time of a request.

5.2 Cost of Checkpoint Manager

The sole cost of the checkpoint manager is that of creating the checkpoint tables. In order to derive an equation that specifies this cost per crash, some additional variables must first be defined as follows:

U = the number of checkpoint entries in the table for the last terminated checkpoint epoch that correspond to objects that were not deleted in that checkpoint epoch

B = the number of checkpoint entries in the table for the last terminated checkpoint epoch that correspond to objects that were deleted in that checkpoint epoch

ΔO = the number of new objects that are created during the average checkpoint epoch

R = the number of VS pages written since the previous crash

P_c = average number of checkpoint entries per VS page that contains at least one checkpoint entry

Using these newly defined variables and those defined in the previous section, the average total cost, C_{cmt}, of the checkpoint manager per crash can be specified:

$$C_{cmt} = [C_{vr}(U-\Delta O + B)/P_c + C_{vr}\Delta O + XC_{or}(U + B) + U/P_c + C_{vw}\Delta O][R/D]$$

Since the updated checkpoint entries are grouped into blocks that occupy a VS page, thereby eliminating the need to write one VS page for every checkpoint table entry that is written, the costs of the VS page reads and writes of these updated checkpoint entries are decreased by a P_c factor. However, since the checkpoint entries for newly created objects are written as the objects are created, it is not possible to group these checkpoint entries into blocks on the VS pages. Thus, the cost of the VS reads and writes of the first checkpoint entry created for every object is not reduced by any pageload factor.

The first two terms, $C_{vr}(U-\Delta O + B)/P_c + C_{vr}\Delta O$, reflect the cost of examining all of the

checkpoint entries in the previous checkpoint epoch table. The third term, $XC_{or}(U + B)$, reflects the cost of examining the corresponding object header for every checkpoint entry in the previous table in order to obtain the current version of the object. The value of X in this term is very close to 1, for the same reason as was given in the previous section. The fourth term, U/P_c, reflects the cost of writing an updated entry for every checkpoint entry that was not deleted in the previous checkpoint epoch. The fifth term, $C_{vw}\Delta O$, reflects the cost of creating new checkpoint entries for newly created objects. This term does not include the cost of reading an OHS page since that cost is attributed to the creation of the object.

The multiplier, R/D, represents the number of checkpoint epochs that exist in VS per crash. Since checkpoint epochs bear no relationship to crash events, this ratio is variable. In other words, checkpoint epochs can be created at any arbitrary rate. Thus, since it is desirable to minimize the repository's response time for satisfying requests, the decision about when to create a new checkpoint epoch will probably be made dynamically by the repository. It will not be a time dependent decision but instead will depend upon D (the distance between the current end of VS and the limiting CEM), and upon the expected usage of the repository.

The decision will depend upon D because the smaller D is, the smaller the values for I_D, V_D, and O_D will be. In other words, the faster new checkpoint epochs are created, the smaller the total cost of the recovery manager will be since the recovery manager will have fewer version images to examine in VS. Nevertheless, this will only decrease the total *response time* cost if object headers get lost due to the crash, since if none are lost then the recovery manager does not scan all the way to the limiting CEM.

However, there is a disadvantage to creating checkpoint epochs at a fast rate: as the rate of creation of checkpoint epochs increases, the ratio, R/D, increases, and therefore, so does the total cost of the checkpoint manager per crash recovery epoch. If the checkpoint manager does its work in the background while the repository is satisfying external requests, the checkpoint epochs should not be created at a very fast rate since the checkpoint manager will be sharing the disk resources with the processes that are handling the external requests, and thus, will increase the repository's response time. However, if the repository has

enough idle time so that the checkpoint manager can do most of its work during that time, then checkpoint epochs can be created at a faster rate since the only cost of the checkpoint manager that will affect the request response time is that of creating checkpoint entries for newly created objects.

Thus, the repository decides to create a new checkpoint epoch if either of the following two situations arise. First, if the repository expects to be idle for some time, the checkpoint manager has finished updating the old table, and some minimum number of new versions have been created in the current checkpoint epoch, then the repository creates a new checkpoint epoch. Second, there is probably some maximum distance over which it is desirable for the recovery manager to ever have to search (because of the time it takes to do all of the necessary disk accesses), so if D reaches half of this maximum, the repository creates a new checkpoint epoch.[17] Thus, the repository creates new checkpoint epochs at the fastest rate that optimizes the repository's time under all conditions.

The parameters specifying the maximum size of D and the minimum number of new versions that should have been created in the current checkpoint epoch will be chosen arbitrarily in the initial implementation of the repository. Then, once it is possible to measure the actual costs and response times of the repository, these parameters will be adjusted.

[17] The reason why the crucial distance is half of this maximum rather than the actual maximum is because the recovery manager has to search through all version images in the previous checkpoint epoch in addition to the current epoch, (the table for the current epoch is not complete until the epoch is terminated).

Since the repository will probably have a reasonable amount of idle time (at least in the wee hours of the morning), the checkpoint manager will do most of its work at that time. The only work that must be done while the repository is satisfying requests is the creation of new object headers. Thus, the average response time cost, C_{cmr}, of the checkpoint manager per crash is:

$$C_{cmr} = (C_{vw}\Delta O)(R/D)$$

One should observe that only a small percentage of the total cost actually affects the repository's response time.

5.3 Average Cost of Recovery Per Request

It would be useful now, to analyze how much the recovery and checkpoint managers cost per request that the repository processes because then we can analyze how these managers affect the repository's response time per request. First, it is necessary to calculate the costs of reading and writing VS and OHS pages.

The costs of VS page reads and writes are:

$$C_{vr} = 1 \text{ disk access} + [\text{page recovery}]^*$$

$$C_{vw} = 4 \text{ disk access} + [\text{repeated diskacceses}]^*$$

Normally, only one disk access is done in order to read a VS page, since only one copy of the page has to be read. However, if a bad VS page is encountered, then there is an additional cost, represented by the term [page recovery], which is the number of disk accesses that must be done in order to recover the page. Since the probability of disk pages decaying is very small, this term will rarely be included in the cost.

In order to write a VS page, at least 4 disk accesses must normally be made, i.e., a read and write for each of the 2 copies of the page that are maintained. However, these 4 disk

accesses represent the *total* cost of a VS write, i.e., the total work that must be done. Since there will probably be two devices performing the writes of both copies in parallel, the *response time* cost of a VS write will only be 2 disk accesses. Furthermore, only in the case where the read back after a write indicates that the write was not done properly and has to be repeated, will the term [repeated disk acesses] become a component cost of a VS page write. Once again, the probability of the original write not succeeding is minimal.

On the other hand, the costs of OHS reads and writes are:

C_{or} = 1 disk access

C_{ow} = 1 disk access

Since OHS is careful (standard disk) storage, each page that is read or written requires only a single disk access.[18]

Now, the average *total* cost of the recovery and checkpoint managers *per request* (excluding all starred terms) is:

$$(C_{rmt} + C_{cmt})/Q = \{I_N/P + V_N + E_N + O_N$$
$$L[I_D/P + V_D + E_D + O_D] +$$
$$[R/D][5\Delta O + (2U - \Delta O + B)/P_c + U + B]\}/Q$$

where Q = the total number of requests satisfied per crash

[18]Note that the actual cost of the OHS read and write operations will be *less than or* equal to 1 full disk access since the OHS page are not read (written) from (to) the disk every time a read (write) is done. Often, the page to be read (written) will be found in a primary buffer. However, if the object header table is big then the reduction in costs will be small.

However the average *response time* cost of the recovery and checkpoint managers *per request* is only:

$$(C_{rmt} + C_{cmr})/Q = [I_N/P + V_N + E_N + L(I_D/P + V_D + E_D) + 2\Delta OR/D]/Q$$

From this equation one can observe how the response time delay that is attributed to recovery fluctuates with the varying characteristics of the requests and objects that are sent to the repository. One thing to notice is that this response time delay decreases as the average size of the clients' objects increse, since the larger the objects are, the smaller the value of V_N and V_D will be. Another thing to notice is that the response time delay increases with the rate of object creation, since the faster new objects are created, the larger the value of ΔO will be.

The following example will give the reader a better feeling for what the actual response time delay that is attributed to recovery per request might be. By choosing an arbitrary but reasonable number of requests that might be processed and a reasonable number of objects that might be valid within a single recovery epoch, approximate values can be extrapolated for all of the terms in the cost equations. Thus, for this example it will be assumed that the repository processes 20,000 requests per crash and that 10.000 objects are current at any given time. The table in Figure 5-1 shows the distribution of request types among the 20,000 requests that are processed and the table in Figure 5-2 shows what values were extrapolated for the variables used in the equations.

Using these values, the average total cost of recovery per request will be:

$$(C_{rmt} + C_{cmt})/Q = 15410 \text{ disk accesses}/20000 \text{ requests}$$

$$= .77 \text{ disk accesses/request}$$

Type	Amount Processed
create-object	1000
delete-object	1000
create-token	5000
read-version	5000
create-comrec	2000
abort-comrec	200
commit-comrec	1750
add-ref	2000
delete-ref	2000
test	50

Figure 5-1: Request Distribution

On the other hand, the average response time cost of recovery per request will be:

$$(C_{rmt} + C_{cmr})/Q = 2050 \text{ disk accesses}/20000 \text{ requests}$$

$$= .1 \text{ disk accesses/request}$$

Thus, in comparison with the average response time costs of processing read-version and create-token requests, which are 2 disk accesses and 1.4 disk accesses, respectively, the additional response time cost attributable to recovery in the normal case, .1 disk accesses, is not very significant.

5.4 Comparative Cost of Another Type of Recovery

To put these costs of recovery into perspective, it is necessary to compare them with similar costs of an alternate method of recovery for the repository. The repository using the recovery mechanisms described in this thesis will be called R, and the alternative will be called R'. Briefly, the design for R' is to implement OHS as reusable stable storage. In R',

Variable	Value
I_D	15000
P_D	5
V_D	5500
E_D	10000
O_D	10000
ΔE	200
U	9000
B	1000
ΔC	1000
R/D	1
P_C	50
N_N	10
V_N	50
I_N	35
E_N	5

Figure 5-2: Extrapolated Values for Variables in Cost Equations

no request is confirmed until the appropriate changes are written into both OHS and VS. Also, all changes made to OHS for a single request are written into OHS from the page buffers in an atomic fashion and are not written until the necessary changes have been made to VS.

Using this alternative design of the repository, it is possible to eliminate the checkpoint manager since object headers will not get lost. Also, the recovery manager can be greatly simplified due to the fact that in fulfilling a request, the repository does not change OHS until VS is modified. Thus, if the repository crashes before updating any part of OHS, then the request will not have been confirmed, OHS will reflect the current state of the data, and the version(s) added to VS will be ignored since the object headers were not changed to include them. In other words, object headers will not become obsolete so there is no need for the recovery manager to search through VS in order to examine the versions and certify

the corresponding object headers.

Therefore, the only responsibility of the recovery manager in R' is to update State Storage before the repository resumes its normal activity. However, since recovery of State Storage is exactly the same for both R and R', its cost will not be included in this comparative analysis. Furthermore, for this analysis it is assumed that the only differences between the two repositories are those that have been described above. Thus, all other costs, such as those for communications, are assumed to be the same in both repositories and will not be included in this analysis.

Superficially, it might appear as if R' uses a more efficient method of recovery. However, the cost of maintaining OHS as stable storage in R' far outweighs the costs of the more explicit recovery mechanisms used in R. This can best be shown, by comparing the costs of satisfying the same types of requests in both repositories (adding the average cost of the recovery mechanisms per request to the cost of satisfying requests in R).

In order to compare these costs, the costs of reading and writing VS and OHS pages in R' must first be calculated. Since there is no difference in the structure of VS for R and R', there is no difference in the costs of reading and writing the VS pages for both repositories. Therefore, C_{vw} and C_{vr} will be used to represent the costs of VS writes and reads for both repositories. (However, for all other costs, any symbols with a prime mark added to them apply to R').

The costs of the OHS read and write operations in R' are greater than those same costs in R. These costs in R' are:

$$C'_{or} = 2 \text{ disk accesses} + [1 \text{ disk access}]^*$$

$$C'_{ow} = 4 \text{ disk accesses} + [\text{repeated disk accesses}]^*$$

An OHS read in R' requires at least 2 disk accesses since OHS is reusable stable storage.[19]

---

[19]Note that in R' the cost of an OHS read will probably be slightly less than 2 disk accesses since the page might be found in the buffer. However, since OHS in R is stable storage, an OHS page has to be written to the disk every time it is modified. Thus, C_{ow} will not be reduced at all.

Thus, both copies of an OHS page must be read and compared, since it is possible for both copies of an OHS page to be valid but different from one another (if the repository crashes in between the writes of the two copies). In this case, where both pages are valid but different, or in the case where one of the pages is bad, one additional disk access is required in order to write the recovered copy of the page.[20]

On the other hand, the cost of the OHS write in R' requires 4 disk accesses because two copies of the page have to be written sequentially and each copy must be read back in order to ensure that the writes were done correctly. However, the term, [repeated disk accesses] is only included in the cost if one of the reads (after a write) indicates that the write was not done correctly and has to be repeated.

Now that the underlying costs of the VS and OHS read and write operations in R' are understood, it is possible to analyze the comparative costs of processing the same type of request in the two different repositories. Two comparisons will be done, one for a create-token request and another for a create-object request.[21] The values from the example in Section 5.3 will be used as the average costs of recovery per request in R. Thus, .77 will be used as the average total cost per request and .1 will be used as the average response time cost per request. In R', there is no additional cost of recovery per request that has to be added into the cost of satisfying a request.

---

[20]Note, that in order to simplify this analysis, the (rare) case where a chain crosses page boundaries is ignored. Thus, it assumed that all buckets in a single chain are fully contained within a single page.

[21]The difference in costs for read-version or delete-object requests is the same as for create-token requests, even though the individual costs differ. Thus, the comparative analysis for these two types of requests will not be done in this thesis.

The average *total* cost of processing a create-token request (assuming that the token fits on a single page) is as follows:

C_{crtkn} = cost of create-token request in R

$= C_{or} + C_{ow} + (C_{vw}/P) +$

average total cost of recovery per request

= 3.57 disk accesses

C'_{crtkn} = cost of create-token request in R'

$= C'_{or} + C'_{ow} + (C_{vw}/P)$

= 6.80 disk accesses

The *total* work that has to be done is less in R than in R'. Furthermore, there is an even greater difference in the average response time costs. In order to obtain the response time cost of satisfying a create-token request in R', the total cost is reduced by half of the cost of the VS page write, since there will most likely be two devices performing the write and read of both copies in parallel. Thus, the response time cost in R' is 6.40 disk accesses. In R, though, the total cost is not only reduced by 1/2 of the cost of the VS page write, but in addition, is reduced by the decrease of .67 in the total recovery cost per request (from .77 to .1 as described in Section 5.3) and by the cost of the OHS page write (which is 1 disk access), since the repository doesn't wait for OHS page writes to complete before responding to requests. Thus, the resulting response time cost of satisfying a create-token request in R is 1.5 disk accesses. This is a significant improvement over the cost of 6.40 disk accesses in R'. Even for a given crash where object headers are lost, the average response time per request would be 2.42, which is still much better than 6.40 for R'.

Next, in the case of a create-object request, the average *total* costs are:

C_{crobj} = cost of create-object request in R

= $C_{or} + C_{ow} + (2C_{vw}/P)$ +

average cost of recovery per request

= 4.37 disk accesses

C'_{crobj} = cost of create-object request in R'

= $C'_{or} + C'_{ow} + C_{vw}/P$

= 6.80 disk accesses

Thus, even though the cost of creating an object in R includes two times the cost of writing a VS page (a checkpoint entry has to be created for the new object in addition to writing the version), the total cost of creating an object in R is less than in R'. Also, there is an even greater difference in the two response time costs since the cost in R drops to 1.90 disk accesses whereas it only drops to 6.40 disk accesses in R'.

Thus, in this example, both the total costs and the response time costs are less for each request satisfied in R than in R'. Even in a rare case where the recovery manager has to recheck all object headers and an additional 2.12 disk accesses must be added to the costs (the starred term in the total cost of the recovery manager, given on page 99), the costs are less in R than in R'. The response time cost of the create-object request, as well as both types of costs of a create-token request are still significantly less in R than in R'.

Note, that R' is not as sensitive to the average size of the objects and the read-version/create-token ratio as R is, nor is it sensitive at all to the rate of object creation, since it does not include a recovery cost term. However, in R, under normal circumstances (where no object headers are lost), the sensitivity of the response time to these variables is still not enough to make the recovery mechanisms in R' more efficient than those in R, with respect to response time.

5.5 Summary

In summary, it has been shown that on the average, although the total cost of these recovery mechanisms is fairly steep, the response time costs of these recovery mechanisms is insignificant. However, it is necessary to keep in mind that these costs are averages. These delays will vary with the requests. The initial requests that arrive after the crash will experience much more response time delay due to the crash than the average delay costs. Nevertheless, once the recovery manager completes its scan, no subsequent requests experience any extra delay due to recovery, except for create-object requests, which require that checkpoint entries be created before the response is sent.[22].

It has also been shown that in the example environment, these recovery mechanisms are more efficient than those used in R', in almost all respects (total and response time costs of all types of requests). Even in the absolute worst case where unassigned or deleted uid's are specified in requests, R is more efficient than R'. It is probable, though, that in an environment where the repository is utilized very heavily, 24 hours a day, and where the objects are fairly large, that R' would provide a more efficient storage service. Although the calculations are only valid for our one example, we have erred in a conservative direction for the example numbers. In general, the recovery cost will probably be less than that in the example.

Finally, if there is any bottleneck in these recovery mechanisms it will be the checkpoint manager since it requires a lot of work to be done just to prevent the worst case from being intolerable. It may have to be made more efficient if certain unfavorable conditions prevail.

---

[22] It can be arranged so that checkpoint entries of newly created objects are written on the same page as the versions of the objects. Then there will not even be any delay attributable to recovery for the create-object requests

Chapter Six

Conclusion

In this thesis, a coherent set of recovery mechanisms for the Swallow repository was presented. In order to sum things up, this final chapter reflects back on the original design goals and then offers suggestions for further work.

6.1 Summary of Original Goals

Recall that the most important goals were to ensure that the repository's data is restored to an internally consistent state and to support the global recovery mechanisms in order to ensure external consistency. The general strategy used to fulfill this goal is to maintain all of the essential data (repository's global state, values of clients' objects and state of the commit records) in stable storage and to restore all auxiliary data from this data in stable storage. Thus, before any auxiliary data is used in order to satisfy external requests, it is always compared with the stable storage data, either explicitly (by scanning sequentially through VS) or implicitly (by comparing the REN's of the repository and the object header), and is brought up to date, if necessary. Furthermore, no data is ever released to external nodes until the state of the corresponding commit record is known to be committed, thus, abiding by and supporting the global recovery mechanisms.

The next goal was to provide minimal disruption to the ongoing activities in the other Swallow nodes by minimizing the immediate recovery that has to be done before the repository can begin accepting requests. The strategy used here is to restore the VS write pointer, the repository's REN and the last uid assigned (to an object or commit record), then to get the checkpoint manager started from where it left off before the crash and finally, to encache the entire global state in volatile storage and start accepting requests. The remaining data, consisting of the object header and commit record tables, are recovered

gradually during the course of the repository's normal activities. Thus, the immediate recovery is trivial.

Of course, even though the repository begins accepting requests fairly soon after a crash, there still may be further delay in returning a response, since the data required to satisfy the request may require recovery. However, the third goal was to minimize this response time delay attributable to recovery. Thus, this goal is met by using non-crash recovery epochs in addition to crash recovery epochs, in order to mark the last point in VS when OHS is guaranteed to be consistent with VS (providing that no object headers are lost). Then, if the repository has frequent idle periods, it will only be necessary to scan through a very small region of VS before a request can be satisfied and confirmed. Furthermore, once that region of VS has been scanned, there will be no additional response time delay attributable to recovery. In other words, all requests will be satisfied at full speed.

6.2 Future Work

The first step that should be taken, now that the recovery mechanisms have been designed, is to use these recovery mechanisms in the repository. Once this is done, the repository's performance can be gauged under various conditions, both normal and stressful, so that all parameters can be fine tuned.

The analysis in Chapter 5 was only intended to give a feel for the costs of recovery. A better analysis could be made by measuring and comparing the actual response time delays of requests arriving immediately after restarting and those arriving some time later. Another interesting measurement would be how the length of time in which the recovery manager performs its required scan through the non-crash recovery epoch preceding the crash recovery epoch varies with different levels of repository utilization. These are only examples of the various anlyses that can be done once actual measurements can be taken.

In addition, the behavior patterns of the users can be monitored in order to figure out what the weaknesses of these mechanisms are. For example, if the repository is more heavily utilized than expected, then the checkpoint and recovery epoch mechanisms may

require modification. However if the usage is as expected, i.e., long periods of idle time during the early morning hours and frequent short periods of idle time through the rest of the day, then these mechanisms should work well.

Another interesting pattern to observe would be the ratio of retransmissions vs. abort-commit-record requests that the repository receives after a crash. If this ratio heavily favors retransmissions then it may be desirable to explore methods for recovering commit records whose final state had not been decided before the crash, other than automatically aborting them.

Finally, new classes of algorithms have been recently developed for hash tables whose size changes dynamically. These algorithms may be incorporated into a subsequent implementation of the object header table in the Swallow repository. If so, then it will be necessary to examine these algorithms for potential difficulties that may be caused by failures and then to modify them so that they can detect and correct any errors before these errors wreak havoc within the repository.

6.3 Generalizations

In a more general sense, the techniques used in the repository for reliably storing, accessing and recovering the data may be applicable to other systems. For example, in the repository, critical data is maintained in stable storage while the optimized mappings to this data are maintained in careful storage. This type of strategy for storing data would be useful in any system that contains some data that cannot be lost. The only deterrent to using this strategy would be the expense of stable storage. Thus, future work should be directed towards reducing the cost of the stable storage read and write operations without decreasing the reliability of the storage.[23]

In addition, the hash table algorithms developed here may lead to convenient methods

[23] In fact, if the stable storage operations could be made sufficiently inexpensive, then there would be no need to have careful storage, at all.

for keeping database indices, since these algorithms are efficient and self-recovering. The essential property of the hash table that allows these algorithms to use trivial error detection and correction procedures is that the hash table does not have to be *perfectly* reliable. In other words, it is acceptable to lose data in the hash table, once in a while. Thus, as long as the hash table data can be recovered from more reliable data sources, if necessary, then a database system can use these algorithms, thereby eliminating the need to check the entire structure of the table of indices for potential damage after a crash, since the hash table algorithms do this check implicitly.

Finally, the notion of online recovery during the normal course of operations is one that would be extremely useful in all computing environments. In order for online recovery to be practical in any given system, cheap methods for detecting the need for recovery as well as for implementing recovery must be developed for that particular system.

In conclusion, there is still work that has to be done in order to fine-tune and perfect the recovery mechanisms within the repository. Even so, these mechanisms can be generalized and applied to other systems in order to improve the standard recovery procedures.

Bibliography

[1]

Accetta, M., Robertson, G., et.al.
The Design of a Network-Based Central File System.
Technical Report CMU-CS-80-134, Carnegie-Mellon University, August, 1980.

[2]

Akkoyunlu, E.S., Ekanadham, K., Huber, R.V.
Some Constraints and Tradeoffs in the Design of Network Communications.
In *Proceedings of the Fifth Symposium on Operating Systems Principles.* ACM, November, 1975.

[3]

Bernstein, P.A., Shipman, D.W., Rothnies, J.B.
Concurrency Control in SDD-1: A System for Distributed Databases; Part 1: Description.
Report CCA-03-79, Computer Corporation of America, Cambridge, Ma., January, 1979.

[4]

Comer, D.
The Ubiquitous B-Tree.
ACM Computing Surveys 11:121-137, June, 1979.

[5]

Gray, J., et al.
The Recovery Manager of a Data Management System.
Research Report RJ2623 (33801), IBM Research Laboratory, San Jose, Ca., August, 1979.

[6]

Israel, J.E., Mitchell, J.G. and Sturgis, H.E.
Separating Data from Function in a Distributed File System.
In *Proceedings of the Second International Symposium on Operating Systems.* IRIA, October, 1978.

[7]

Knuth, D.E.
The Art of Computer Programming - Sorting and Searching, Volume 3.
Addison-Wesley Publishing Company, 1973.

[8]
Lampson, B. and Sturgis, H.
Crash Recovery in a Distributed Data Storage System.
Xerox Palo Alto Research Center, Ca. April, 1979. To appear in CACM.

[9]
Lindsay, B.G., et. al.
Notes on Distributed Databases.
Technical Report RJ2571 (33471), IBM Research Laboratory, San Jose, Ca., July, 1979.

[10]
Maurer, W.D., Lewis, T.G.
Hash Table Methods.
ACM Computing Surveys 7( ):5-19, March, 1975.

[11]
Paxton, W.H.
A Client-Based Transaction System to Maintain Data Integrity.
In *Proceedings of the Seventh Symposium on Operating Systems Principles.* ACM, December, 1979.

[12]
Postel, J.
User Datagram Protocol.
Technical Report IEN-88, USC-Information Sciences Institute, May, 1979.

[13]
Randell, B., Lee, P.A., Treleaven, P.C.
Reliability Issues in Computing System Design.
ACM Computing Surveys 10(2):123-165, June, 1978.

[14]
Reed, David P.
Naming and Synchronization in a Decentralized Computer System.
PhD thesis, M.I.T., September, 1978.

[15]
Reed, D.P.
Implementing Atomic Actions on Decentralized Data.
Presented at the Seventh Symposium on Operating Systems Principles sponsored by ACM. To appear in CACM.

[16]
Reed, D.P., Svobodova, L.
Swallow: A Distributed Data Storage System for a Local Network.
Presented at International Workshop on Local Networks sponsored by IBM Zurich Research Laboratory in August, 1980.

[17]
Svobodova, L.
Reliability Issues in Distributed Information Processing Systems.
In *Proceedings of the Ninth IEEE Fault Tolerant Computing Symposium*, pages 9-16. IEEE, June, 1979.

[18]
Svobodova, L.
Management of Object Histories in the Swallow Repository.
Technical Report MIT/LCS/TR-243, M.I.T., July, 1980.

[19]
Swinehart, D., McDaniel, G., Boggs, D.
WFS: A Simple Shared File System for a Distributed Environment.
In *Proceedings of the Seventh Symposium on Operating Systems Principles.* ACM, December, 1979.

[20]
Verhofstad, J.
Recovery and Crash Resistance in a Filing System.
In *Proceedings of the ACM-SIGMOD Conference on Management of Data.* ACM, August, 1977.

SECURITY CLASSIFICATION OF THIS PAGE *(When Date Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MIT/LCS/TR-252	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: Recovery of the Swallow Repository	5. TYPE OF REPORT & PERIOD COVERED: S.M.Thesis - Jan.1981
	6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TR-252
7. AUTHOR(s): Gail C. Arens	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0661
9. PERFORMING ORGANIZATION NAME AND ADDRESS: MIT/Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: ARPA/Department of Defense, 1400 Wilson Boulevard, Arlington, VA 22209	12. REPORT DATE: January 1981
	13. NUMBER OF PAGES: 122
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: ONR/Department of the Navy, Information Systems Program, Arlington, VA 22217	15. SECURITY CLASS. *(of this report)*: Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

This document has been approved for public release and sale; its distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

distributed data storage system
hash table
recovery
optical disk computer system reliability

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This thesis presents the design of a set of recovery mechanisms for the Swallow repository. Swallow is a distributed data storage system that supports highly reliable long term storage of arbitrary sized data objects with special mechanisms for implementing multi-site atomic actions. The Swallow repository is a data storage server that keeps permanent data in write-**once stable storage such as optical disk.**

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE