Reliable Object Storage to Support Atomic Actions

by

Brian Masao Oki

May 1983

© Massachusetts Institute of Technology 1983

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract number N00014-83-K-0125, and in part by the National Science Foundation under grant number 82-03486MCS.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Reliable Object Storage to Support Atomic Actions

by

Brian Masao Oki

Submitted to the
Department of Electrical Engineering and Computer Science
on May 19, 1983 in partial fulfillment of the requirements
for the Degree of
Master of Science in Computer Science

Abstract

To preserve the consistency of on-line, long-lived, distributed data in the presence of concurrency and in the event of hardware failures, it is necessary to ensure atomicity and data resiliency in applications. The programming language Argus is designed to support such applications. This thesis investigates the mechanism needed to support the notion of data resiliency present in Argus. Data resiliency means that the probability is very high that the crash of a node or storage device in a distributed system does not cause the loss of vital data. Data resiliency requires the use of stable storage devices, memory devices that survive failure to a high probability. This thesis is not concerned with how to implement stable storage devices, but rather with *how to organize the use* of stable storage. The thesis presents a new organization of stable storage called the *hybrid log* that provides fast writing of information to stable storage and reasonably fast recovery of information from stable storage. In the context of this scheme, various algorithms are developed for writing objects to the log, recovering objects from the log, and housekeeping the log.

Thesis supervisor: Barbara H. Liskov
Title: Professor of Computer Science and Engineering

Keywords: Atomic actions, atomic objects, distributed systems, logs, recovery, shadowing, stable storage, transactions

Acknowledgments

First and foremost, I thank Barbara Liskov, my thesis supervisor, for the numerous and always enlightening conversations I had with her concerning my thesis, and for patiently reading and editing my oftentimes verbose and godawful drafts.

I thank my office mates, Jim Restivo (Restivo-person), Jennifer Lundelius (Lun delirious), and Sheng-Yang Chiu who tolerated me while I researched and wrote this thesis. I thank Mr. Ng (I know, mister is not a verb)--Tony Ng--who argued with me over some of the finer points of ideas and forced me to think about them more carefully. Dave Gifford not only read my thesis proposal some time ago and suggested possible references, but also helped me in numerous other ways. Flaviu Cristian, Sheldon Finkelstein, and C. Mohan of the IBM San Jose Research Laboratory read parts of my thesis and provided useful comments. Bruce Lindsay, also of IBM Research, carefully read my thesis and pointed out an oversight in my discussion of two-phase commit, which I have corrected. Jim Gray of Tandem Computers also read my thesis and encouraged me.

Thanks are due to the members of the Programming Methodology Group (secretly known as the Distributed Systems Group) for listening to my ideas and suggesting helpful ways of improving them. Bill Weihl, Chiu, and Maurice Herlihy read early notes I had written and commented extensively. Paul Johnson and Bob Scheifler patiently answered my incessant stream of questions and deciphered the mysteries of some of the infamous DSG notes in a way that I could understand.

I extend my sincere thanks to those people not connected with this thesis, but who nevertheless touched my life in some way. Navy Lieutenant Dillard George kept me from breaking a leg and imperiling the lives of fellow skiers on the "Wildfire" slope of Killington's Bear Mountain. Asa Simmons, former co-worker at Hughes Aircraft Company-Fullerton and now a manager at Raytheon in Rhode Island, invited me to stay with him and his wife over the Patriot's Day weekend in 1982 when I most needed to get away. Ziyad Duron, my erstwhile roommate, tolerated my idiosyncrasies. Jeannette Wing and Alok Vijayvargia patiently listened to my ramblings about various things that bothered me. Judy Zinnikas and I shared our thesis woes.

I thank my good friends, Kevin Wallace, Rob Granville, Bob Gounley, and Richard Sproat, who have, in their own way, made life here at M.I.T. more bearable for me. Kevin and Rob, fellow computer science graduate students, provided a sympathetic ear for my numerous complaints and problems. Space enthusiast Bob Gounley was present at a certain event in April, 1982 that did much to relieve my apprehensions; I am grateful for that and numerous other kindnesses. Silly linguist Richard Sproat (said he, "Stable Storage, or How to Put Your Horse Away") saw me through some trying emotional times over the past year.

I thank my parents, who provided love, gentle encouragement, and guidance throughout my academic career.

Most of all, my heartfelt thanks go to my wonderful little sister, Lisa, whose boundless faith and unflagging pride in me managed to keep me going even when I started to lose faith in myself. Although she was 3000 miles away in California, I still felt her presence strongly and saw her reassuring smile in my mind's eye.

Table of Contents

Chapter One: Introduction 7

1.1 Stable Storage 8
1.2 Organizing Stable Storage 9
1.2.1 Logging versus Shadowing 9
1.2.2 The Approach 11
1.3 Related Work 12
1.3.1 System R Recovery Manager 12
1.3.2 Swallow 13
1.4 Outline of thesis 14

Chapter Two: Background 16

2.1 The Programming Language Argus 16
2.2 Two-phase Commit Protocol 18
2.2.1 The Coordinator 18
2.2.2 The Participant 19
2.2.3 Effects of crashes on Two-phase commit 19
2.3 The Recovery System 20
2.4 Recoverable Objects 22
2.4.1 Atomic Objects 22
2.4.2 Mutex Objects 23
2.4.3 Incremental Copying Algorithm 23

Chapter Three: Simple Log -- Writing and Recovery Algorithms 25

3.1 Log abstraction interface to stable storage 25
3.2 Structure of the simple log 26
3.3 Writing objects to the log 29
3.3.1 The Coordinator 29
3.3.2 The Participant 29
3.3.3 Writing data entries 30
3.3.3.1 Copying Data 30
3.3.3.2 What to Write 32
3.3.3.3 The Writing Algorithm 39
3.4 Recovering objects from the log 40
3.4.1 Sketch of the General Algorithm 41
3.4.2 Log Scenarios 42
3.4.3 Turning uids into pointers 50
3.4.4 The General Recovery Algorithm 51

Chapter Four: Hybrid Log -- Writing and Recovery Algorithms 54

4.1 Simple log versus Hybrid log 54

4.2 Writing objects to the log 55
4.3 Recovering objects from the log 57
4.3.1 Sketch of the General Algorithm 57
4.3.2 Log Scenario and Recovery 58
4.3.3 The General Recovery Algorithm 59
4.4 Early prepare 60

Chapter Five: Hybrid Log -- Housekeeping Algorithms 63

5.1 Compacting the log 64
5.1.1 The Compaction Algorithm 65
5.1.2 The New Recovery Algorithm 68
5.2 Taking a snapshot of the stable state 69
5.3 Summary 71

Chapter Six: Conclusions 73

References 75

Table of Figures

Figure 1-1: Shadowed objects 11
Figure 2-1: The Recovery System 20
Figure 2-2: An Atomic Record 24
Figure 3-1: Data entries and Outcome entries 27
Figure 3-2: Format of recoverable objects in volatile memory 30
Figure 3-3: Objects in volatile memory 31
Figure 3-4: Flattened Object 32
Figure 3-5: Newly Accessible Objects Example 34
Figure 3-6: Newly Accessible Objects 38
Figure 3-7: Log of atomic objects after a crash 42
Figure 3-8: Log of mutex objects following a crash 45
Figure 3-9: Log following a crash 46
Figure 3-10: Coordinator's log following a crash 48
Figure 4-1: New format of log entries 56
Figure 4-2: Log after the prepare phase 57
Figure 4-3: Hybrid log after T1 prepares and T2 commits 62

Chapter One

Introduction

In applications such as banking systems, airline reservation systems, office automation systems, and database systems, the manipulation and preservation of long-lived, on-line, distributed data is of primary importance. The Argus programming language and system [Liskov 82], currently under development at M.I.T., is designed to support such distributed applications. A major issue in such applications is preserving the consistency of on-line data in the presence of concurrency and in the event of hardware failures. One aspect of this issue is the problem of how on-line data can be made resilient to hardware failures, which means that the probability is very high that the crash of a node or storage device does not cause the loss of vital data. This thesis is concerned with supporting data resiliency in Argus.

To maintain data consistency in the presence of concurrency it is necessary to make the activities that use and manipulate the data *atomic*. Atomic activities are referred to as *actions* or transactions [Davies 73, Davies 78, Eswaran 76]. An atomic action is *indivisible* and *recoverable*. Indivisibility means that the execution of one action never appears to overlap the execution of any other action. Recoverability means that the overall effect of an action is all-or-nothing, that is, either all changes made to the data by the action happen (the action *commits*), or none of these changes happen (the action *aborts*).

To support data resiliency, Argus causes data to be written to stable storage devices when an action commits. These devices are special memory devices that have a high probability of surviving failure [Lampson 79]. If an action aborts, on the other hand, then changes the action made to the data are discarded.

This thesis investigates the mechanism that supports Argus's notion of data resiliency. There are two aspects to providing resiliency: implementing stable storage devices and organizing the use of stable storage. We are concerned with the latter: how stable storage can be organized in an efficient way that allows a distributed computer system to recover

from failures.

In this chapter, we begin by explaining the notion of stable storage, and then we discuss issues that arise in the use of stable storage. We discuss two possible organizations, logging and shadowing, and briefly sketch our approach. Then we discuss related work and provide an outline of the remainder of the thesis.

1.1 Stable Storage

Stable storage is similar to other kinds of memory storage except that it has the property that it is much less likely to fail. We can imagine that stable storage looks like a conventional magnetic disk in that it provides the usual read and write operations; the write operation, however, is *atomic*, which means that the data is either written completely to the disk or not written at all, even if there is a failure while the update is happening. This atomicity property ensures that the data will never be left in an inconsistent state in which the old value is gone and the new value is wrong. We call this kind of stable storage *atomic stable storage*.

Since this ideal storage device whose properties we have described does not exist, it must be implemented using conventional storage devices with less desirable properties. Lampson and Sturgis [Lampson 79] describe a method for implementing atomic stable storage using magnetic disks. The basic idea is to use two disk pages to represent each page of data that must be updated atomically, and to update one and then the other. These two disk pages reside on different physical storage devices that have completely independent failure modes, that is, the failure of one storage device should not influence the other storage device. The associated storage management overhead for stable storage is comparatively more expensive than for conventional storage because of the extra memory and I/O involved in maintaining a second copy of the data and the time needed to update the second copy. This extra expense has an impact on how stable storage is used and the manner in which it is organized.

In this thesis we are not concerned with the issues and the manner in which stable storage is implemented. Instead, we assume that atomic stable storage exists, has the right properties, and is available to use; this stable storage forms the basis for the reliable object

storage organization presented in the thesis.

1.2 Organizing Stable Storage

The main issue in organizing stable storage is deciding what data structures to use. Influencing the decision are three requirements: writing data objects to stable storage, recovering data objects from stable storage, and efficiency. The efficiency requirement attempts to strike a balance between the writing and the recovering requirements. This section discusses two main methods for organizing stable storage and then briefly outlines our approach.

1.2.1 Logging versus Shadowing

There are two possible approaches to organizing stable storage: the *pure log scheme* and the *shadowing scheme*.

Like an accounting journal that is a chronological record of accounting transactions, a *log* is a kind of stack data structure that grows at one end as information is written onto it. It is used to record the object values that were changed by an action as well as the outcomes of the action [Bjork 75, Gray 78, Lampson 79]. The outcome information is needed to recognize when a crash occurs before all values of data objects modified by a committing action are written to the log. On recovery, all modifications for such an action will be discarded (and the action aborts).

If stable storage is organized as a log, then writing is fast because information is written only to the end of the log. On the other hand, recovery from a crash tends to be slow because the entire log must be consulted to restore the state.

The traditional notion of *shadowing* involves writing new versions of the objects changed by an action to stable storage without writing over the previous object versions (called the shadowed object versions, following [Gray 78]); two versions of each object can exist in stable storage. When the new object versions have been completely written, these newly written objects supplant the previous object versions in one atomic step. Thereafter, all references to the objects will access the new object versions.

Storage is organized as a pointer to a table, called the *map*, which associates object

identifiers (uniquely identifying each object) with the actual objects in stable storage. As an action gets ready to commit, the object versions are written to stable storage. When an action actually commits, these new versions are installed by making a new map that contains the pointers, writing the map to stable storage, and then switching from the old map to the new map in one atomic step. (This switch is accomplished by updating the pointer to the map.) The action is now committed and its changes have been reflected in the map as well as in stable storage. Any references to objects now pass through the new map. When an action aborts, the new object versions are discarded and the map is untouched. Figure 1-1 shows what such a scheme might look like.

After a crash, the map is consulted in order to restore the objects because the objects to be recovered are easily accessible in the map.

If the data an action manipulates is distributed, then a map alone is not enough for shadowing to work properly. A log is also required. The log contains entries for each action that was in the process of committing but that had not yet either committed or aborted. Objects modified by an action are written to stable storage and pointers to these objects are placed in a log entry. When an action truly commits, these changes are installed in the map; if the action aborts the map remains untouched and the changes are discarded. Committing, then, occurs in two phases. First, the objects are written to stable storage and the pointers to the modified objects are written to an entry in the log. Second, the map is updated with these pointers. This is similar to the two-phase commit protocol, which we discuss in the next chapter.

After a crash, the map and the log are both consulted to restore the objects. The map represents those objects that were modified by a committed action. The log represents those actions that had not yet either committed or aborted and whose changes had been written to the log but had not been installed in the map.

The advantage of shadowing is that recovery is fast. The disadvantage of shadowing involves changing the entries in the map and rewriting the map at every action commit, which could be expensive, especially if the map is large and there are a large number of objects.

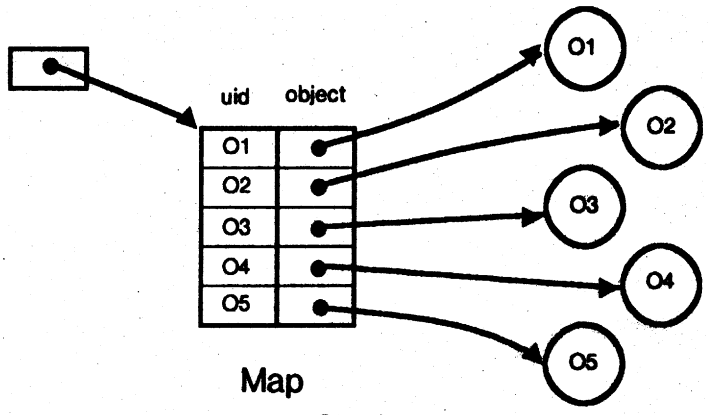


Figure 1-1: Shadowed objects

1.2.2 The Approach

Let us summarize the advantages and disadvantages of these two schemes:

1. Log ⇒ fast writing, but slow recovery
2. Shadowing ⇒ slow writing, but fast recovery

In comparing the two approaches we assume that crashes do not happen very often and that we would like normal processing to be fast at the possible expense of a slow recovery after a crash.

For reasons to be discussed in later chapters, we have chosen an organization of stable storage that falls between these two extremes, which we call the *hybrid log*. As the name suggests, it is a hybrid of the pure log and the shadowing schemes that combines the advantageous characteristics of either scheme. Hence, writing is almost as fast as the pure log, and recovery is faster than the pure log scheme but not quite comparable with the shadowing scheme. The map in the shadowing scheme is now written incrementally to the hybrid log and is distributed over the entire log; this means that the extra cost associated with updating the map at every action commit in the shadowing scheme is just part of the cost of writing entries to the log.

Given this hybrid organization, we have also developed three kinds of algorithms: (1) writing objects to the hybrid log, (2) recovering objects from the hybrid log, (3) and

reorganizing the hybrid log to make recovery from crashes more efficient.

In summary, this thesis's contribution is two-fold:

1. the hybrid log organization for stable storage, and
2. the various algorithms for writing, recovering, and housekeeping the hybrid log.

1.3 Related Work

There are several systems that are related to the work described in the thesis. Two of these systems, the System R Recovery Manager and Swallow, are discussed below.

1.3.1 System R Recovery Manager

Developed at the IBM Research Laboratory in San Jose, System R [Astrahan 76] is an experimental relational database management system designed to support transaction processing. It uses shadow-paging and a log protocol to implement recovery for committed and uncommitted actions. We will briefly discuss the recovery mechanism of System R [Gray 81].

In System R, all information is stored on files, some of which are shadowed files. A buffer manager maps these files into a virtual memory buffer pool that is volatile and does not survive node failures. For the purposes of this section, we are interested in shadowed files. Shadowed files consist of two versions, a *current* version and a *shadow*, or backup, version. Any operations affect only the current version and never touch the shadow version, except when the current version becomes the new shadow version due to a file SAVE command or when the shadow version is used to restore the current version due to a file RESTORE (which undoes recent updates). Note that the current version of a shadowed file is volatile and goes away after a node crash, but the shadow version survives. Following a node crash, shadowed files are reset to their shadow versions.

A problem that System R designers encountered was that it was not clear how to generalize the shadowed files (current version/shadow version) technique to transactions running concurrently on a shared file. They desired to commit or to undo changes *per transaction*, which was not possible using shadowed files alone. Suppose several transactions were running concurrently and made changes to a shadowed file. When it was

time to commit or abort the updates, file save or restore would commit or abort the updates of *every* transaction that altered the file, even those that had not yet committed and might abort later. Because of this problem, the designers chose to combine the shadow mechanism with an incremental log of all actions that a transaction performed.

To ensure that committed transactions can be redone and that uncommitted transactions can be undone, System R does the following:

1. The transaction log is forced to disk *before* the current database state replaces the shadow database; this is called the *write ahead log protocol* [Gray 78].

2. A transaction commits at the moment its commit record appears on disk. The commit log record is written to the buffer and then all the transaction's log records are forced to disk.

System R designers realized fairly late in their implementation that the transaction log made the shadow mechanism redundant for large shared files. Shadowing was expensive and more complex, and appeared to be bad for direct processing and sequential processing. They felt that much of the I/O that went on was inherent in the shadow mechanism and probably could not be eliminated altogether [Gray 81].

In short, System R designers first looked at the shadowing mechanism. Realizing that the shadowing mechanism itself was inadequate and would not allow them to commit and undo changes on a per transaction basis, they combined it with the log mechanism to deal with this problem: the log was used not only to record changes made to data items per transaction, but also to undo changes on recovery. Thereafter, they recognized that the log made shadowing unnecessary and that the log was all they really needed.

We think that the fundamental problem in System R was the mismatch in the sizes of two things: the unit of recovery--the page--was different from the unit of synchronization--a record on a page. If the sizes had matched, a simpler scheme would have been possible. Even so, their conclusions about the inefficiency of shadowing remain valid.

1.3.2 Swallow

The work most closely related to the hybrid log organization of stable storage and the recovery system is Swallow [Reed 81], developed at M.I.T. It is an "integrated system of servers that provides reliable, secure, and efficient storage for clients throughout a

network" [Arens 81] where a server is a node (a computer or a network) that provides resources and a client is a node that shares and uses the resources. Swallow is intended for a distributed computer system. Among other things that it provides is extremely reliable storage for client data through the Swallow *repository*, so that the likelihood of losing client objects is very small.

The Swallow repository supports Version Storage (VS) as the main form of atomic stable storage. VS contains the versions of objects and commit records (indicating the state of the transaction). VS looks very much like a log and is always increasing in size, so only a portion of it remains on-line. The repository manages VS in such a way that the current versions of objects and commit records always remain on-line [Svobodova 80].

Like a shadowing scheme, the repository maintains a separate table called the Object Header Table, which is kept in secondary storage; there is just one copy of the table and modifications are made in place. Each object has an object header in the table that, among other things, points to the most current version of the corresponding object in VS; in this way, the repository need not sequentially scan through VS just to find the versions of objects. Arens points out that the table is a hint; these object headers are not necessary for the repository to function correctly, because it can always resort to searching through VS, but that they are necessary for the repository to function efficiently.

1.4 Outline of thesis

Chapter 2 presents background information concerning the Argus programming language and model of computation, the notion of atomic actions, the recovery system, recoverable objects, and the two-phase commit protocol. This chapter sets the stage for the technical exposition of Chapters 3, 4, and 5.

Chapter 3 presents the writing and recovery algorithms in the context of the simple log. In particular, we discuss the structure of the log, the format of log entries, and determining accessibility of objects. We then illustrate the recovery algorithm through four scenarios.

Chapter 4 builds on the work presented in Chapter 3. In particular, we argue that the hybrid log organization of stable storage is better in general than either the pure log or the

shadowing schemes considered alone. We then explain the writing and recovery algorithms for the hybrid log. Finally, we point out the complications introduced by the notion of early prepare.

Chapter 5 considers the problem of reorganizing the hybrid log to make recovery from crashes more efficient. Two methods are discussed and compared: log compaction and stable state snapshot.

Finally, in Chapter 6 we summarize the foregoing, draw conclusions, and suggest directions for further research.

Chapter Two

Background

This chapter lays the groundwork for understanding the remaining chapters. We first explain the basic concepts underlying Argus, which supports the writing of distributed programs and serves as the context in which this thesis research was done. Argus provides consistency and reliability through atomic data and the mechanisms of actions and a two-phase commit protocol. Next, we discuss a two-phase commit protocol, which ensures that the effects of a successful atomic action are made permanent and guarantees that everyone involved in the action either commits or aborts; data objects modified by the action are written to stable storage during two-phase commit. We then consider the recovery system, which is the interface between the Argus system and stable storage. The job of the recovery system is to write these data objects to stable storage as needed, to restore the data objects after a crash, and to reorganize stable storage in order to make recovery from a crash more efficient. Finally, we introduce *recoverable objects*, which are certain data objects that are written to stable storage.

2.1 The Programming Language Argus

Argus [Liskov 82] gives programmers the ability to write distributed programs that run on a network of computers. Each node in the network is an independent computer consisting of one or more processors and some local memory; each node may differ in the number and types of processors, the amount of memory, or in attached peripheral devices; and each node can communicate with the others only by sending messages over the network.

In Argus, a distributed program consists of modules called *guardians*. A guardian encapsulates and controls access to resources, such as databases or devices, and guards its local data. A guardian's external interface is in the form of a set of operations, called *handlers*, that can be called by other guardians to provide access to the called guardian's

objects. In addition to the data objects, there are processes in a guardian that perform background tasks and execute the handler calls.

Guardians are the logical nodes of the distributed system, and each resides at a single physical node, although a node may support several guardians. Guardians survive crashes of their nodes of residence with high probability. When a guardian's node crashes, all processes within the guardian disappear, but a subset of the guardian's state survives. A guardian has both *stable* state and *volatile* state: the stable state survives crashes of a guardian's node and the volatile state disappears. The stable state consists of objects called *stable objects*; these objects are accessible[1] from the guardian's stable variables, which are variables whose declarations in a guardian definition are prefixed by the keyword **stable**. Only stable objects survive crashes. To ensure that these objects really do survive, they must be recorded with care on stable storage devices, which we mentioned in the last chapter. When the guardian's node recovers from a crash, the Argus system re-creates the guardian with the stable objects as they were when last written to stable storage. A process is then started in the guardian to initialize the volatile objects. Once the volatile objects have been restored, the guardian can resume background tasks and can respond to new handler calls.

Atomic actions are the primary method of performing distributed computations in Argus. Actions are atomic: the effect is all-or-nothing, that is, they either complete successfully (commit) and change the current state permanently, or fail completely (abort) and restore the state that existed before the action was executed. In Argus, an action called a *top-level action* starts at one guardian and can spread to other guardians, spawning subactions by means of handler calls. When an action completes, it either commits at all guardians (and the changes made by the action to each guardian's stable state are reflected in stable storage appropriately) or aborts at all guardians.

---

[1] In Argus, variables refer to objects, and objects may refer to other objects. The objects accessible from a variable are those that the variable refer to, and those referred to by objects accessible from the variable.

2.2 Two-phase Commit Protocol

As actions execute, modifications of objects are made to volatile copies. When a top-level action has completed its work and wishes to commit, we need some way of making sure that the modifications made by the various subactions on behalf of the top-level action either are all written to stable storage, in which case the top-level action really commits, or are discarded, in which case the top-level action aborts. The standard *two-phase commit protocol* fulfills this need [Gray 78]. The protocol works even if crashes occur while it executes. In the explanation that follows, we assume that no nodes crash forever and eventually any two nodes can communicate [Moss 81]. Following standard terminology, let us call the guardian where the top-level action executes the *coordinator*, and the guardians where the data was modified by the top-level action or its subactions the *participants*.

2.2.1 The Coordinator

Preparing phase

In the preparing phase, the coordinator sends a *prepare* message to all participants saying "prepare for action A to commit," where A is the action identifier of the preparing action, and then waits for the participants to respond that either they are prepared or they have aborted.

After sending out *prepare* messages to all the participants (including itself if it was also a participant), the coordinator waits for replies. If it hears from each participant that each has prepared it starts the committing phase. If any participant replies *aborted*, then the coordinator tells the participants to abort via *abort* messages. The coordinator may also abort unilaterally if it does not receive responses from some participants; the Argus system determines when such an abort should occur. For example, if a participant has crashed it clearly cannot respond, and in this case the coordinator will unilaterally abort the action.

Committing phase

If all participants respond *prepared*, the coordinator creates a *committing* record and writes it to stable storage. At this point the action is committed. The coordinator then sends *commit* messages to all the participants (including itself), informing them of its verdict, and

waits for them to respond. When all have responded *committed* the coordinator creates a *done* record and writes it to stable storage. Two-phase commit is now complete.

2.2.2 The Participant

Prepare phase

When a participant receives a *prepare* message from the the coordinator for some preparing action it prepares in the following way. First, the data records for all objects modified by the preparing action are written to stable storage. Second, if the data records were written successfully to stable storage, then a *prepared* record is written to stable storage. The participant then replies *prepared* to the coordinator, and enters the completion phase. If the action is unknown at the participant (because it never ran there, was aborted locally, or was wiped out by a crash), then the participant replies *aborted* to the coordinator.

Completion phase

Once a participant has written the *prepared* record, it must await the verdict from the coordinator. When the participant receives the *commit* message, it writes a *committed* record to stable storage, and then replies *committed* to the coordinator. If the participant receives the *abort* message instead, it writes an *aborted* record to stable storage. If a participant has not heard from its coordinator it can query the coordinator to find out the outcome of the action. (The action id contains enough information such that each participant knows who its coordinator is.)

2.2.3 Effects of crashes on Two-phase commit

If a participant crashed before the *prepared* record was written to stable storage for some preparing action, then all record of that action is lost, and the action will be aborted. When the *prepared* record appears in stable storage the action is really prepared. This entry marks the point of no return for the participant; after this point, the participant must wait for the coordinator to inform it of the outcome. If a participant crashed after the *prepared* record was written to stable storage, then the recovery system restores the guardian's state as it had been before the crash, when the action had prepared and was waiting to hear from the coordinator.

If a coordinator crashed before the *committing* record was written to stable storage for some committing action, then it will remember nothing about the action after recovery, and the action will be aborted. If the coordinator receives a query about the action from a participant, it will tell the participant to abort the action. When the *committing* record appears in stable storage the action has really committed; this entry marks the point of no return for the coordinator, after which it must commit. Suppose, however, a coordinator crashed after the *committing* record was written to stable storage, but before the *done* record was written. Then upon recovery the action is still committing and the recovery system restores the guardian's state as it had been before the crash.

If a coordinator crashed after the *done* record was written to stable storage for some committing action, then this action has completed and nothing special need be done.

2.3 The Recovery System

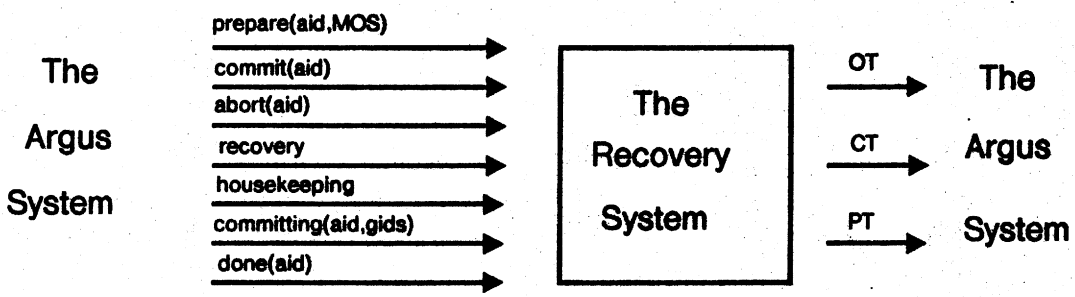


Figure 2-1: The Recovery System

The job of the recovery system is to write information to stable storage as needed by two-phase commit, to restore a guardian's stable state after a crash, and to reorganize stable storage in order to make recovery more efficient. The recovery system provides operations that the Argus system calls at appropriate times in order to carry out these tasks. See Figure 2-1. The Argus system itself is distributed, every guardian containing a portion of it; the recovery system also exists at each guardian and is called by the portion of the Argus system at that guardian.

Not all objects at a guardian need to be written when an action commits. In particular, objects not accessible from the stable variables need not be written. If some previously existing stable object was not modified then its correct copy already exists on stable storage and need not be copied. A modified stable object, however, must be copied to stable storage, as must a newly created stable object. To minimize writing to stable storage, the Argus system keeps track of just the modified objects and newly created objects. The Modified Objects Set (MOS) contains this information; it consists of two smaller sets, the set of objects modified by an action and the set of objects newly created by the same action.

Let us discuss the functions of the recovery system in terms of the calls made on it by the Argus system.

1. *prepare(aid, MOS)*. The Argus system at the participant's guardian calls this operation when it receives a *prepare* message for the action *aid* from the coordinator. The first argument is the action identifier, and the second argument, MOS, is the Modified Objects Set, which we have explained above. Not all objects in the MOS are written to stable storage; only those that are accessible from the stable variables are written to stable storage because they are the ones that make up the guardian's stable state. Each accessible object in the MOS is written to stable storage on behalf of this action, the prepare record is written, and then prepare returns. The Argus system at this participant's guardian then replies *prepared* to the coordinator.

2. *commit(aid)*. When the Argus system at the participant's guardian receives a *commit* message from the coordinator, it calls this *commit(aid)* operation, which writes the commit record to stable storage and then replies *committed*.

3. *abort(aid)*. The Argus system at the participant's guardian calls this operation when it receives an *abort(aid)* message from the coordinator. The recovery system writes the abort record to stable storage and returns; the Argus system then replies *aborted*.

4. *committing(aid, gids)*. When the coordinator has heard that all participants have prepared, the Argus system at the coordinator's guardian calls the *committing(aid, gids)* operation; the first argument is the *aid* of the committing action and the second argument is a list of identifiers of each guardian that was involved in the action. The recovery system writes the committing record to stable storage and returns; the coordinator enters the second phase of two-phase commit.

5. *done(aid)*. When all participants have responded *committed* to the coordinator, the Argus system at the coordinator's guardian calls this operation to indicate that two-phase commit is over and the action has terminated. The recovery system writes the done record to stable storage and returns.

6. *recovery*. Following a crash, the Argus system calls the *recovery* operation of a guardian's recovery system to restore the stable state of that guardian. After the recovery system has restored the guardian's stable state, it returns information to the Argus system at the guardian. The returned information is sufficient to resume any running actions at that guardian. We will explain what this information is (OT, CT, and PT) in the next chapter.

7. *housekeeping*. Whenever the Argus system has determined that enough old information has accumulated on stable storage at a guardian, it calls the *housekeeping* operation to reorganize the stable storage to make recovery more efficient.

In this thesis, we assume that these operations are called sequentially by the Argus system.

2.4 Recoverable Objects

A guardian's stable objects are grouped into special objects called *recoverable objects*. These are the units that are written to stable storage. They come in two flavors: *built-in atomic objects* and *mutex objects*.

2.4.1 Atomic Objects

Argus provides an assortment of built-in atomic objects. These objects are similar to ordinary objects except that they are implemented in a way that provides atomicity for actions that use them. Providing atomicity means synchronizing the using actions and providing recovery for the using actions. These are provided by read/write locks and volatile versions in volatile memory.

The Argus implementation of built-in atomic objects is based on a fairly simple locking model. There are two kinds of locks: read locks and write locks. To use an object, an object must invoke an object operation; the operation acquires a lock in the appropriate mode and the action holds the lock until it completes. When a write lock is obtained, a *version* of the object is made (in volatile memory), and the action operates on this version. If the action ultimately commits, this version will be retained and the old version discarded. If the action aborts, this version will be discarded, and the old version retained.

At two-phase commit, the committing action will still hold write locks on atomic objects that were modified and the atomic object has two versions: the *base* version and the *current*

version. If the object were previously accessible, then only the current version will be copied to stable storage (the base version already appears on stable storage). For newly created atomic objects, the creating action holds a read lock on the object and there may be only a single version, the *base* version, which is copied to stable storage.

2.4.2 Mutex Objects

The mutex object is like a container for other objects and has a lock associated with it. Actions that wish to modify the mutex object must first gain possession by executing a *seize* operation (an Argus language construct); the seize operation provides mutual exclusion for the action in possession.

Mutex objects have only one version, namely, the current version, and it is this version that gets written to stable storage. Mutex objects also have a different semantics from atomic objects at the prepare phase. Once an action has prepared at a participant's guardian, all mutex objects take on their new states even if the action aborts later; after a crash, these new states must be restored even if an abort record for the action has been recorded. See the paper by Weihl and Liskov [Weihl 82] for more details on mutex.

2.4.3 Incremental Copying Algorithm

The method we use for copying recoverable objects to stable storage has the following properties [Weihl 82]: it is incremental and order-independent. The algorithm works in an *incremental* fashion: each built-in atomic object and mutex object is written to stable storage in a separate, atomic step. In copying each such object, the system copies all portions of the object except contained atomic and mutex objects. These are copied separately if they were modified or are new. The algorithm is also *order-independent*: the atomic and mutex objects are written to stable storage in an arbitrary order.

For each recoverable object that was modified the recovery system copies the object in the following fashion:

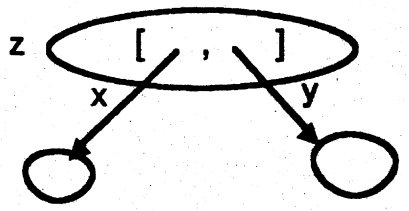
1. The recovery system gains possession of the object. This is really of interest only for mutex objects. In this case, the recovery system seizes the mutex lock to synchronize the copying with the user code.

2. The recovery system copies all non-recoverable objects contained in the

recoverable object but not any contained recoverable objects; these will be copied separately if they were modified. The sharing of objects is preserved only for shared recoverable objects or for a group of unrecoverable objects entirely contained within a recoverable object.

3. Once the recoverable object, including its contained non-recoverable objects, has been copied, the recovery system releases possession and continues.

To copy a recoverable object, the system invokes a routine that linearizes (or flattens) the data in the modified object and in any contained non-recoverable objects. Any references to other recoverable objects are translated from their volatile addresses to their corresponding stable storage references. Figure 2-2 illustrates this technique. In copying the object referred to by variable z, we copy x but not y (since y is atomic but x is not); instead, we place a stable storage reference for y in the copy of z, and copy y separately if necessary (if it was modified or was new).



z: atomic record[x: int, y: atomic array[int]]

Figure 2-2: An Atomic Record

In short, the system gains possession of each recoverable object that had been modified by the action, copies it, releases possession, and continues.

Chapter Three

Simple Log -- Writing and Recovery Algorithms

This chapter focuses on the algorithms for writing recoverable objects to the log when a top-level action commits and for recovering these objects from the log and restoring them to volatile memory upon recovery from a crash. We ignore the issue of housekeeping until Chapter 5.

We begin by discussing an abstraction, called the stable log, that serves as the interface to stable storage and whose operations the recovery system calls whenever objects are to be written to stable storage. Then we discuss a simple way of using the stable log. We call this method the *simple* log to distinguish it from a more involved variation presented in the next chapter. The simple log is made up of data entries and outcome entries, and we present the format of each kind. Next, we cover the algorithm for writing recoverable objects to the simple log in the context of two-phase commit and discuss not only how objects are written but what objects are written. Finally, we present the full-fledged algorithm itself as well as several recovery scenarios to illustrate the recovery algorithm in action.

As mentioned in the last chapter, we assume that recovery system operations are executed sequentially.

3.1 Log abstraction interface to stable storage

In Chapter 1, we imagined that stable storage devices might resemble conventional magnetic disks. These disks provided the usual read and write operations, but the write operation had the property that it was atomic: the object updates were either written completely or not at all. We also mentioned that several techniques were available to implement stable storage but that we were not concerned with these techniques.

Even this pseudo-disk interface is inappropriate for our purposes. Instead, we postulate the existence of a stable storage system that provides objects that look like stable

logs and behave like stable logs. These stable log objects provide the right interface to the recovery system and are presumably implemented in an efficient way that again does not concern us. Each guardian has its own stable log. The stable log looks like an array indexed by abstract objects called log_address. (From now on we will use the term "log" to mean stable log.)

The stable log abstraction provides the following operations [Raible 83]:

1. *write(log,entry)*. This operation writes conveniently sized blocks, called entries, to the stable log object. The actual writing of the data to the stable storage device may not have happened when this operation returns.

2. *force_write(log,entry)*. This operation forces an entry to the log object. Both the current entry and any older entries that were not yet written to the stable storage device will be written by this operation before it returns.

3. *read(log, log_address)*. Given the log object, this operation reads the entry at the log address and returns it.

4. *read_backward(log, log_address)*. Given the log object, this operation reads the log backwards starting at the specified log address, one entry at a time, and returns each entry.

5. *get_top(log)*. Given the log object, this operation returns the log address of the last entry that was forced to the log.

6. *create()*. This operation creates a new log object and returns it.

7. *destroy(log)*. This operation destroys an existing log object.

3.2 Structure of the simple log

As explained in the previous chapter, a *log* is a data structure like a stack that grows in one direction as information is appended to it. There are two kinds of log entries that contain this information: *data entries* and *outcome entries*. Data entries contain the information about recoverable objects that needs to be recorded on stable storage; outcome entries indicate the *outcomes* of actions, that is, whether an action has prepared, committed, or aborted.

The log as described here is actually used for two distinct purposes: (1) recording data, and (2) recording action states of the participant and of the coordinator.

As shown in Figure 3-1, a data entry consists of four fields: (1) the unique identifier

(*uid*) of the recoverable object, (2) the object type--mutex or atomic, (3) the object value, and (4) the action identifier (*aid*) of the top-level action that is preparing. The object "value" is not the actual object itself residing in volatile memory but a *copy* of the object's version.

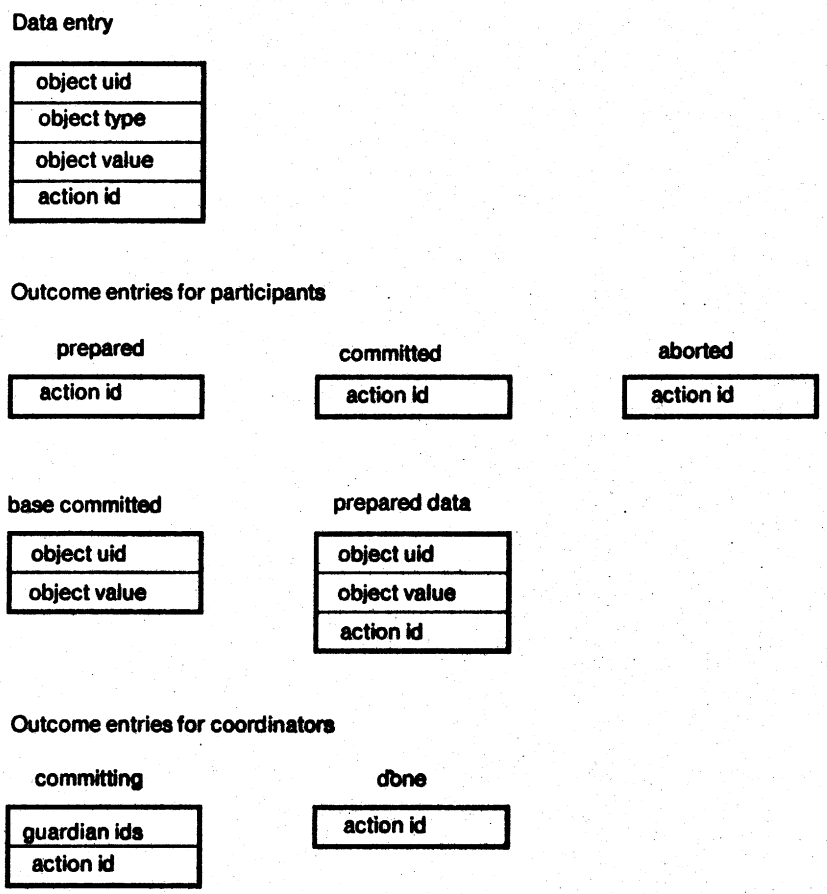


Figure 3-1: Data entries and Outcome entries

The object's unique identifier is some identifier that will never be reused and is unique with respect to the object's guardian. Since this identifier will not serve any other purpose except to distinguish recoverable objects from one another, the unique object generator can be a *stable counter* associated with each guardian, that is, an integer that is incremented whenever a recoverable object needs a uid. There is no danger of a uid being reused after a crash because the recovery system knows after recovery of each guardian the last uid that was generated and assigned to a recoverable object at that guardian; the stable counter can

be reset and new uids generated from that point. (We assume that uids are big enough that it is highly unlikely that we would run out.)

The action identifier is generated by the Argus system in some fashion, which does not concern us. We assume that it is given.

Notice in Figure 3-1 that outcome entries come in two varieties, one set for participants and the other set for coordinators. At certain distinct points during two-phase commit a participant is in one of three states:

1. *prepared* if it received a *prepare* message from the coordinator and successfully wrote data entries to its associated guardian's log.

2. *committed* if it received a *commit* message from the coordinator.

3. *aborted* if it received the *abort* message.

A participant is said to be in one of these three states only after the *prepared*, *committed*, or *aborted* outcome entry is written to the log by the participant.

In addition, there are two special participant outcome entries, *base_committed* and *prepared_data*, that handle certain cases that arise when object versions are written to the log. The *base_committed* entry contains the object uid, object type, and object value; the *prepared_data* entry has almost exactly the same format as a data entry (missing the object type). We will explain the purpose of these entries in later sections. For now, we note that these entries are like combined data and outcome entries. When one of these special entries is written, it is akin to writing not only the data entry, but also a *prepared* outcome entry (in the case of *prepared_data*) or a *prepared* outcome entry followed by a *committed* outcome entry (in the case of *base_committed*).

Coordinators can be in one of two states during two-phase commit: (1) *committing* if all participants in the action have prepared themselves, and (2) *done* when all participants responded *committed*. A coordinator is said to be in the *committing* state after the *committing* outcome entry appears in the log and is in the *done* state after the *done* outcome entry is written successfully to the log.

Notice that the coordinator is really just like a participant except that it performs some extra tasks, namely, appending these coordinator outcome entries to the log. Moreover, depending on the role a guardian plays during two-phase commit, the guardian can be

either a coordinator or a participant; thus, a guardian's log could contain outcome entries for a coordinator when the guardian acts as coordinator and for a participant when the guardian behaves like a participant.

We will elaborate further on these different outcome entries in the next several sections when we discuss the writing of objects to the log.

3.3 Writing objects to the log

Recoverable objects are written to the log *only* when top-level actions commit and to ensure that effects of top-level actions are made permanent, the system goes through the standard two-phase commit protocol described in the previous chapter.

3.3.1 The Coordinator

After sending out *prepare* messages to all the participants (including itself since it is also a participant), the coordinator waits for replies. If any participant replies *aborted*, or if the coordinator aborts unilaterally, then the coordinator tells the participants to abort via *abort* messages. If it hears from each participant that each has prepared it starts the committing phase.

If all participants respond *prepared*, the recovery system creates a *committing* outcome entry and forces it to the coordinator's log. (Whenever we say that a log entry is forced to the log, we mean that the *force_write* operation on the log object is invoked with the log entry.) At this point the action is committed. The coordinator then sends *commit* messages to all the participants (including itself), informing them of its verdict, and waits for them to respond. When all have responded *committed* the coordinator creates a *done* coordinator outcome entry and forces it to the coordinator's log. Two-phase commit is now complete.

3.3.2 The Participant

When a participant receives a *prepare* message from the the coordinator it prepares in the following way. In general, for each object in the MOS the recovery system constructs data entries and writes them to the log. If the data entries were written successfully to the

log, then the recovery system forces a *prepared* outcome entry to the log. The participant then replies *prepared* to the coordinator. If the action is unknown at the participant (because it never ran there, was aborted locally, or was wiped out by a crash), then it replies *aborted* to the coordinator.

When the participant receives the *commit* message, the recovery system creates a *committed* outcome entry and forces it to the participant's log, and then the participant replies *committed* to the coordinator. If the participant is told to abort, then the recovery system creates an *aborted* outcome entry, and forces it to the participant's stable log.

The writing of data entries is discussed in detail in the next section.

3.3.3 Writing data entries

3.3.3.1 Copying Data

Figure 3-2 shows how recoverable objects are formatted in volatile memory.

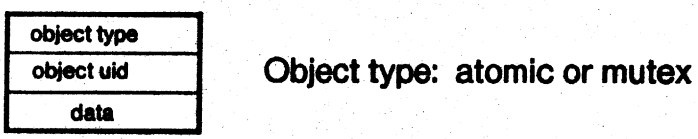


Figure 3-2: Format of recoverable objects in volatile memory

There are three kinds of objects in the Argus world as far as the recovery system is concerned: atomic objects, mutex objects, and regular objects. These are distinguished by their "type" field in the object. Regular objects lack the object uid field.

The uid field in objects makes sense for atomic and mutex objects because these objects are assigned unique identifiers. The data field is the actual object itself, which may include volatile references to other objects. If the object is atomic then this data field will consist of at most two versions during two-phase commit; if the object is mutex then this data field will be just the current version.

How are these objects residing in volatile memory written to the log? To copy a recoverable object, the recovery system invokes the incremental copying algorithm

discussed in Chapter 2 on the data portion of the recoverable object, in particular, on the appropriate version (current or base version if the object is atomic, or the current version if the object is mutex). As the copy proceeds, the algorithm follows volatile memory references, replacing references to recoverable objects with their uids and simply copying any regular objects. The data is now flattened. The recovery system then creates a data entry containing the object uid, the action id of the action that is preparing, the object type, and the flattened data. And it is this data entry that is written to the log.

Figure 3-3 shows a possible situation involving atomic, mutex, and regular objects.

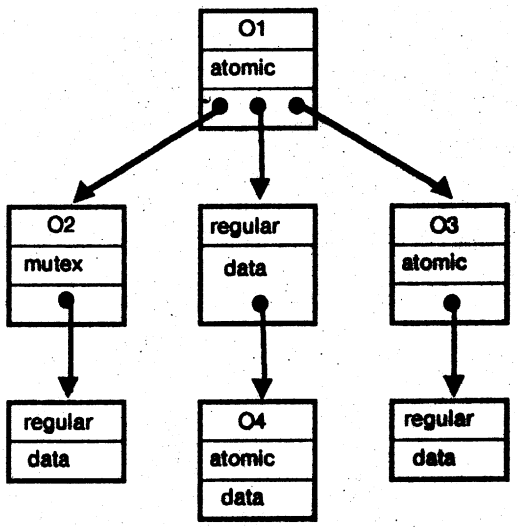


Figure 3-3: Objects in volatile memory

Suppose object O_1, which was modified by action T_1, is to be copied to the log. The incremental copying algorithm follows pointers in the data portion of the object. The reference to object O_2 (a mutex object) is replaced with the uid O_2 itself. The algorithm copies the regular object and in so doing discovers that it contains a reference to yet another recoverable object, namely O_4, an atomic object; it replaces the reference with O_4 itself. And finally, the algorithm replaces the reference to object O_3, an atomic object, with the uid O_3.

In flattened form, O_1 looks like Figure 3-4.

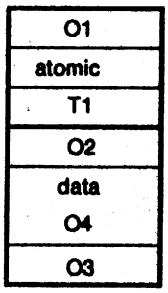


Figure 3-4: Flattened Object

3.3.3.2 What to Write

Having discussed the manner in which data is copied to the log as data entries, let us consider the question of what actually gets written. As we mentioned before, we are interested only in those recoverable objects that are accessible from the stable variables because these make up the stable state of the guardian and only the stable state survives crashes. Recall that, for each action, the Argus system keeps track of both modified objects and newly created objects in the MOS and does not distinguish between objects accessible from the stable variables and objects accessible from the volatile variables. It is the job of the recovery system, then, to separate the objects in the MOS that are accessible from the stable variables from those objects that are inaccessible and to write the accessible objects to the log.

Notice that this concern with accessible objects is really an optimization because we could simply write out *all* the recoverable objects at a guardian without regard for accessibility or inaccessibility; if some inaccessible object were written out to stable storage it would not matter since it was unreachable anyway, but it would clutter the log with irrelevant information.

The Problem of Newly Accessible Objects

Recoverable objects are either previously accessible from the stable variables or newly accessible.

Let us consider previously accessible recoverable objects. If the previously

accessible object is a built-in atomic object, then in the prepare phase of two-phase commit the action still holds the necessary write lock on the object (it was granted a write lock in order to modify the object), which prevents any other action from concurrently modifying the object. The recovery system can then copy the object's current version. For mutex objects, since only one version exists, that version is copied after the recovery system seizes the mutex lock. In either case, the recovery system copies the version, creates a data entry containing the object uid, the object type (mutex or atomic), the copied version, and the action id, and writes the data entry to the log.

Let us consider newly accessible recoverable objects. If an action made some object accessible that was not accessible before, the object is said to be *newly accessible* and no version of it appears in stable storage at all. The only way to make these recoverable objects accessible is to modify some already accessible object. How do previously inaccessible objects come about? They come about when an object is newly created or was previously reachable only from a volatile variable.

Newly accessible objects cause a problem that is illustrated by the following scenario as depicted in Figure 3-5. Let us examine a guardian's stable state as two different actions modify it. In the figure, O_1, O_2, and O_3 are atomic objects; and X and Y are stable variables: X points to object O_1 and Y points to object O_2. Here, we have simplified the format of objects somewhat, in order to emphasize the presence of different versions. For atomic objects, a single box means the base version and that an action holds either no lock or a read lock; a double box means both current and base version and that an action holds a write lock. For mutex objects, a single box means the current version. "Int" means any regular object that is neither atomic nor mutex.

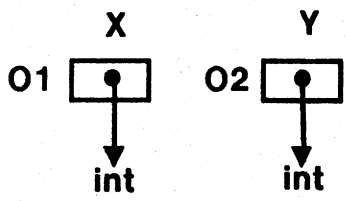
We look at how the initial situation shown in step 1 is transformed into the situation shown in step 8.

1. No action holds a lock (read or write) on any recoverable object in the initial situation. Action T_1 had modified objects O_1 and O_2 and had committed.

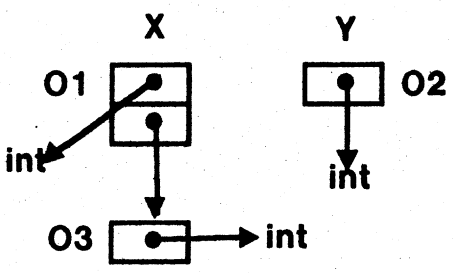
2. Action T_2 obtains a write lock on O_1, creating a new version. The action creates a new atomic object O_3 with a read lock. This new version is modified to point to object O_3.

3. Action T_3 obtains a write lock on O_2, creating a new version. This version of O_2 is modified by the action to point to O_3, which is also pointed to by O_1.

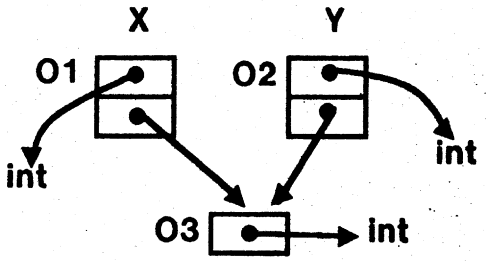
Figure 3-5: Newly Accessible Objects Example



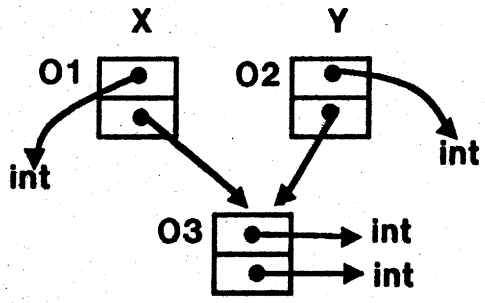
1. Initial situation



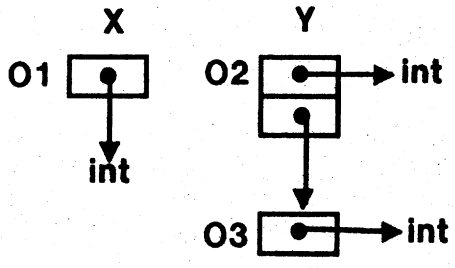
2. T2 modifies O1 -> O3



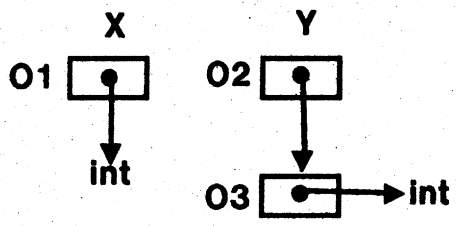
3. T3 modifies O2 -> O3



4. T2 modifies O3
5. T2 prepares
6. T3 prepares



7. After T2 aborts



8. After T3 commits

4. Action T_2 obtains a write lock on O_3 and modifies it.

5. Action T_2 prepares.

6. Action T_3 prepares.

7. Action T_2 aborts.

8. Action T_3 commits.

9. The Argus system crashes.

The point of this example is this: Even though T_2 aborted, object O_3 must be recovered after a crash because it is needed for T_3. To ensure survival of a newly accessible object if it is a mutex object is no problem. The recovery system simply creates a data entry containing the object uid, the object type mutex, the copied mutex version, and the action id. This solution is sufficient because the mutex object will be restored from this version even if the action that is preparing aborts later. For atomic objects, however, this version would be discarded if the preparing action later aborted. To avoid this, we write out the *base* version of the object using a special outcome entry, *base_committed*, containing the object uid, and the copied object version.

Actually, the discussion above is slightly simplified. Suppose action A makes an object newly accessible, and that object had been modified by some other action, B, and B has already prepared. When B prepared, the object was not copied to the log since it was not accessible. Therefore, in preparing A, we must write out the object's *current* version as well as its base version. The current version is needed in case B commits; the base version is needed in case B aborts. Therefore, in the case where the newly accessible object is an atomic object, and this object is write-locked by an action *that has prepared*, the recovery system creates, in addition to the *base_committed* outcome entry, another special outcome entry, the *prepared_data* entry, containing the object uid, the copied current version, and the action id of the modifying action. The recovery system writes this entry to the log.

To enable the recovery system to recognize prepared actions, it maintains for each guardian an internal table called the *Prepared Actions Table* (PAT) that contains action ids of actions that are prepared. This table finds its use in the situation described above: when the recovery system encounters a newly accessible object that is write-locked by some other

action, we ask whether the modifying action had prepared by looking up its action id in the PAT; if the answer is yes, then we copy both the base and current versions; if the answer is no, then the recovery system just copies the base version. When an action commits or aborts, its action id is removed from the PAT.

Notice that the objects stored in *base_committed* and *prepared_data* outcome entries are always atomic, so the object type is not really needed in these entries. This is why the object type is not included in those entries.

Notice that newly created objects must always be newly accessible. They need not be included in the MOS because they would be discovered through the mechanism described in the next section. Therefore, we modify the MOS to contain just the objects that were modified by an action.

The Accessibility Set

Thus far, we have assumed that the recovery system somehow knew which recoverable objects were previously accessible and which recoverable objects were newly accessible, and we described how these objects were dealt with when written to the log. The recovery system maintains, for each guardian, an *accessibility set* (AS) that enables the recovery system to know which recoverable objects at a guardian are previously accessible and which are newly accessible.

We define the accessibility set as a set that contains the uids of those objects that are known to be accessible from a guardian's stable variables. Why an accessibility set? What is wrong with the recovery system walking the graph of recoverable objects from the stable variables in order to determine which objects are accessible? If it reaches some object from a stable variable, then surely the object must be accessible. The problem is that traversing the graph can be potentially expensive, especially since the recovery system is obliged to follow pointers. The accessibility set (AS) is an attractive alternative because it is a simple matter (and relatively cheap) to look up an identifier and ask whether the object to which it refers is accessible: (1) if the uid is in the set, then the object is accessible from the stable variables and should be copied to the log; (2) if the object uid is not in the set, then either the object is not accessible, or the object is newly accessible and should be copied to the log.

Accessibility is based on a guardian's stable variables, so it is necessary for the

recovery system to know about these variables. For the purposes of this thesis, we can imagine that the guardian's stable variables are collected in a single, recoverable object with a predefined uid. This object contains the association between each stable variable and the uid of the recoverable object directly accessible from it. We assume that this object is created with its uid when the guardian itself is first created. The recovery system knows the uid, so that on recovery after a crash it can initialize the stable variables to their recoverable objects.

Processing the MOS Using the Accessibility Set

Figure 3-6 shows a situation involving a newly accessible object. We illustrate how a prepare message is processed using the accessibility set.

Stable variable X points to atomic object O_1, which in turn points to atomic object O_2. The accessibility set consists of uids O_1 and O_2; O_3 is not in this set because it is not accessible from X. Suppose action T_1 was granted a write lock on object O_2, thus creating a new version (figure a), and the action modified O_2 to point to object O_3, another atomic object (figure b). O_3 is now newly accessible since it was not previously accessible. The action holds a read lock on this object. Action T_1 prepares at this guardian. In the call of the recovery system prepare operation are two arguments, the action id T_1 and the MOS containing O_2. What does the recovery system do?

1. An empty newly accessible object set (NAOS) is created. This set will contain the objects that the recovery system determines are newly accessible.

2. The recovery system checks the accessibility set (AS) for the object uid O_2 and finds that it is present, and therefore the object is indeed accessible. The current version should be copied to the log.

3. As the current version is copied, the recovery system finds a reference to another recoverable object, O_3. It checks the AS for the object uid, does not find it, and inserts the object into the NAOS. This set is processed after the recovery system has processed every object in the MOS.

4. The recovery system creates a data entry that contains the object uid O_2, the object type atomic, the copied version, and the action id T_1 and writes the entry to the log.

5. The recovery system has finished processing the MOS (consisting of object O_2), and now considers the NAOS consisting of object O_3. Since the object is in this set the recovery system knows the object is newly accessible and must be

treated differently. Since the object is an atomic object that the action has a read lock on (and thus there is only a single version), the recovery system creates an outcome entry, *base_committed*, consisting of object uid O_3, and the copied object version. The recovery system writes the entry to the log, deletes object O_3 from the NAOS, and inserts uid O_3 into the AS.

6. The NAOS is empty, so the recovery system is done. It has determined which of the objects in the MOS were accessible and has written the corresponding data entries to the log. It forces a *prepared* outcome entry to the log.

7. The AS now consists of object uids O_1, O_2, O_3.

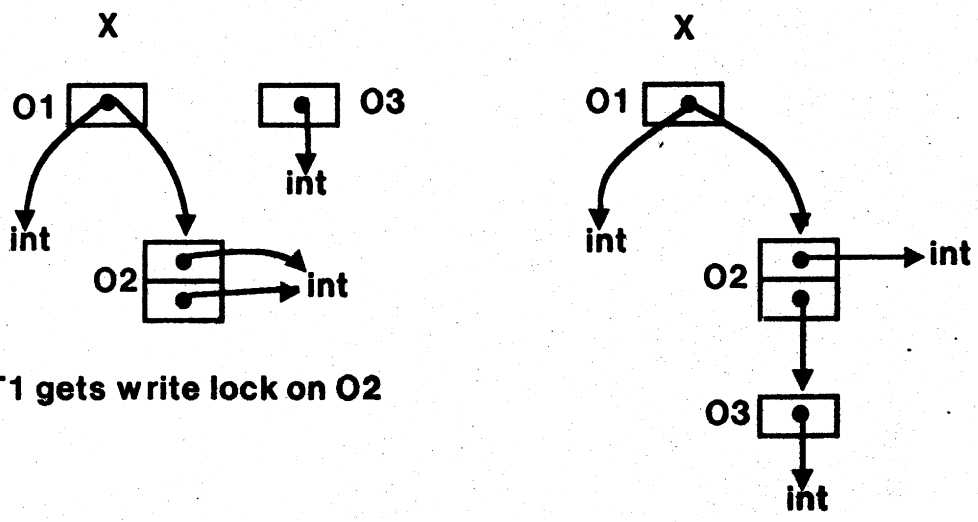


Figure 3-6: Newly Accessible Objects

Notice that there are two phases. First, the recovery system processes every object in the MOS (which was one of the two arguments in the call of the prepare operation), copying current object versions and writing data entries to the log as it goes along. As these object versions are copied, recoverable objects not previously accessible (that is, their uids are not already in the AS) may be revealed as newly accessible; these objects are placed in another set, the NAOS, consisting of just newly accessible objects.

Second, when the recovery system has processed the MOS, it then proceeds to process the NAOS, if it is not empty. After each object is processed it is deleted from the NAOS and added to the AS. Other recoverable objects may become newly accessible and

are added to the NAOS, so that they, too, will be processed by the recovery system. This procedure continues until the NAOS is empty, which means that the recovery system is done writing the data entries on behalf of some preparing action, and can write the *prepared* outcome entry to the log.

As actions execute at some guardian and modify objects, they may make recoverable objects that were once accessible from the stable variables inaccessible. Their uids continue to remain in the accessibility set and so the set grows larger over time. The accessibility set, then, can be thought of as a kind of superset of the recoverable objects that actually constitute a guardian's stable state in the sense that it may contain inaccessible objects in addition to the accessible objects of the stable state. If the set grows too large, then the set should be trimmed. The recovery system would start up a process in parallel with normal processing at the guardian and traverse the recoverable objects accessible from the stable variables. During the traversal it would fill an empty AS with the object uids of accessible recoverable objects. When the process has completed its task it intersects the new set with the old set to yield a new set, which is the new accessibility set. The reason the two sets are intersected instead of the new one supplanting the old is that during the traversal newly accessible objects may be included in the new AS; they should be eliminated from the new AS because newly accessible objects are treated differently and the intersection accomplishes this.

3.3.3.3 The Writing Algorithm

Let us summarize the algorithm for writing recoverable objects to the log as data entries. We assume that a participant has received the *prepare* message from the action's coordinator.

1. Create an empty NAOS.
2. Check the AS. If AS is empty, then the recovery system fetches those objects directly accessible from the stable variables and inserts them into the NAOS. (This situation arises only when a guardian has just been created.)
3. $\forall$ object $\in$ MOS do
    a. Check the AS.

b. If object uid $\in$ AS, and therefore the object is accessible, then the recovery system copies the object's current version. The recovery system creates a data entry that contains the object uid, the object type (atomic or mutex), the copied object version, and the action id. The entry is written to the log. As the object version is copied, the recovery system flattens the data and checks the AS for every recoverable object it comes across. If uid $\notin$ AS then the object is newly accessible and is inserted in the NAOS.

c. If the object uid $\notin$ AS then ignore it.

4. $\forall$ object $\in$ NAOS do

a. If this action has a write lock on the object then both the base version and the current version are copied. For the base version the recovery system creates a *base_committed* entry containing the object uid, and the copied base version and writes the entry to the log. For the current version, the recovery system creates a data entry containing the object uid, object type, copied current version, and action id and writes the entry to the log.

If this action has a read lock on the object then the recovery system creates a *base_committed* entry containing the object uid, and copied current version and writes it to the log.

If this action has no lock, but some other action A holds a write lock, then the recovery system checks the PAT for A. If A $\in$ PAT then that action has prepared, and both base and current versions must be copied. The base version is copied as before; a *prepared_data* outcome entry is created and contains the object uid, copied current version, and action id A of the modifying action. If A $\notin$ PAT or if there is no other action A holding a write lock on the object, then the recovery system copies the base version as above; the *base_committed* entry is written to the log.

b. Insert the object uid in the AS and delete the object from the NAOS.

c. As the object version is copied, new recoverable objects may become accessible. They are added to the NAOS and will eventually be processed.

5. The participant forces the *prepared* outcome entry to the log and then responds *prepared* to the coordinator.

3.4 Recovering objects from the log

After a crash, the recovery system reads the log backwards starting with the last outcome entry, constructs tables to help keep track of action states and objects it has seen, and reconstructs each recoverable object in volatile memory. When it has finished, the

stable state of the guardian has been restored.

We first sketch the algorithm to emphasize the important aspects and to give the reader an intuitive idea of what the algorithm does. Having sketched the algorithm, we present four log scenarios, each chosen to illustrate certain situations that can arise and the manner in which the algorithm deals with them. The first scenario concerns only built-in atomic objects and their versions. The second scenario concerns the complications to the algorithm introduced by mutex objects. The third scenario shows how newly accessible recoverable objects are handled. The fourth scenario shows a guardian that acts both as a coordinator and as a participant. We then explain how uids are turned into volatile memory references. Finally, we present the full-fledged algorithm for recovering objects from the log.

3.4.1 Sketch of the General Algorithm

1. Create three tables: (1) an object table (OT) that maps object uids to both object states (*prepared* or *restored*) and object locations in volatile memory, (2) a coordinator action table (CT) that maps action ids to coordinator action states (*committing* and *done*, where *committing* also has a list of guardian ids of guardians involved in the action), and (3) a participant action table (PT) that maps action ids to participant action states (*prepared*, *committed*, and *aborted*).

2. Read the log *backwards*, starting with the last outcome entry in the log. For every log entry, process it in the following way.

    a. Fill the three tables with appropriate information (action ids and action states like *prepared*, *committed*, and *aborted*).

    b. If necessary, copy the appropriate object version to volatile memory.

3. Make a final pass over volatile memory, replacing object uid references with volatile memory references. Once this pass is completed, all recoverable objects have been reconstructed, and the guardian's stable state has been restored.

4. Create a new, empty AS, and traverse the stable state, filling the AS with the uids of those objects that are actually accessible.

5. Return these three tables to the Argus system in order to resume the tasks of the participants and coordinators.

3.4.2 Log Scenarios

Scenario 1--atomic objects

Suppose the situation depicted in Figure 3-7 exists at a participant's stable log after a crash.

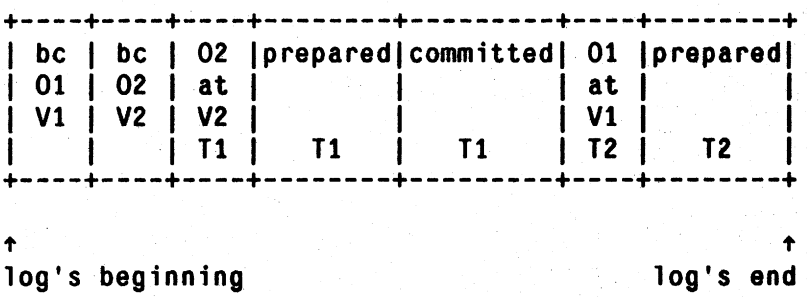


Figure 3-7: Log of atomic objects after a crash

In this figure (and all figures of this sort) the beginning of the log is on the left and the end of the log is on the right; the log grows to the right. The symbols in the log depicted have the following meaning. T_1 and T_2 are actions. Action T_1 has committed; action T_2 has prepared. O_1 and O_2 represent unique object identifiers; and V_1 and V_2 are the object values, that is, the *versions* of objects.

Let us develop some notation to make it easier to talk about data entries and outcome entries in a log. Let data entries be represented as quadruples:

<object uid, object type, object version, action identifier>

so a data entry might look like $\langle O_1,\text{atomic},V_1,T_1\rangle$, where O_1 is the object uid, atomic indicates that the object version is atomic, V_1 is the object version, and T_1 is the action id. Let us represent outcome entries as doubles of

<outcome, action identifier>

and so the first two outcome entries would look like $\langle\text{prepared},T_1\rangle$ and $\langle\text{committed},T_1\rangle$. The only exception is *committing*, which also includes a list of guardian ids. Furthermore, we represent the special outcome entries in the following way:

<bc, object uid, object version>

where "bc" is short for *base_committed*;

<pd, object uid, object version,action id>

where "pd" is short for *prepared_data*.

The system builds two tables: (1) a participant action table (PT) for remembering the action identifiers of actions that either prepared, committed, or aborted.

PT: action id → participant action state

where participant action state = {prepared, committed, aborted}; (2) an object table (OT) maps uids of objects the system has seen to the object state-- either *prepared* if the action that modified the object had prepared but not committed or *restored*--and volatile memory addresses. These are addresses of the objects that were restored to volatile memory.

OT: object uid → object state + vm address

where object state = {prepared, restored} and vm address is a volatile memory address. The objects recorded in the object table will be precisely those that were written by actions that either prepared or committed and whose action identifiers appear in the PT.

In reconstructing the stable state of the guardian, the recovery system begins reading the log backwards, one entry at a time, starting with the last outcome entry appended to the log before the crash (it invokes the *get_top* operation on logs to get the log address of the last outcome entry). The following is a step-by-step explanation of how the objects in this log are recovered; each step refers to a particular log entry, so step 1 explains the very last log entry, step 2 the second to the last log entry, and so on to the beginning of the log.

1. For outcome entry <prepared, T_2>, the recovery system enters the action id T_2 into the PT, and sets the associated action state to *prepared*. The state for the action id T_2 is *prepared*, which means that the action had successfully prepared during two-phase commit but had not yet been told to either commit or abort.

2. For data entry <O_1,atomic,V_1,T_2> the recovery system checks the PT for the action id and the associated action state. The state is *prepared*. The recovery system checks the OT for the object uid, O_1, finds that it is not present, enters the uid into the OT, and sets the object state to *prepared*. The recovery system copies V_1 into volatile memory and sets the vm address to the object address in volatile memory; it represents the current version of the object. What this object state *prepared* says is that the latest *committed* version of this object must be copied to volatile memory as well. Since the action held a write-lock at the time of the crash, the action is granted a write-lock on the object.

3. For outcome entry <committed,T_1>, the recovery system enters the action identifier, T_1, into the PT, setting the associated action state to *committed*.

4. For outcome entry <prepared,T_1> that the recovery system next considers, it consults the PT once again to check the action state associated with the action identifier; since the state is already *committed*, the recovery system ignores the entry.

5. For data entry $\langle O_2, atomic, V_2, T_1 \rangle$ the recovery system consults the PT, noticing that the action id T_1 is in the PT and that the action state is *committed*. The object uid does not appear in the OT, and so the recovery system enters the uid O_2 into the OT, copies V_2 into volatile memory as the current version for an atomic object, sets the object state to *restored*, and sets the vm address to the object address in volatile memory.

6. For outcome entry $\langle bc, O_2, V_2 \rangle$ the recovery system knows that V_2 of object O_2 is a base version (it is a *base_committed entry*); it checks the OT, finds that the current version of the object has already been copied to volatile memory, and ignores the entry.

7. For outcome entry $\langle bc, O_1, V_1 \rangle$, the first log entry, the recovery system knows this is *base_committed* entry and that V_1 is a base version. The object uid, O_1, already exists in the OT with state *prepared*. The recovery system copies V_1 into volatile memory as the base version for an atomic object and resets the object state in the OT to *restored*.

8. Finally, the recovery system makes a pass over the objects in volatile memory to translate any object uid references to volatile memory references. We will elaborate on this later. The stable counter is reset to O_2.

9. The recovery system creates an empty AS, traverses the stable state in volatile memory from the stable variables, and fills the new AS with the uids of accessible objects. In this case, the new AS contains uids O_1 and O_2, assuming that O_1 and O_2 are accessible.

10. The PT and OT tables are returned to the Argus system in order to resume the participant.

At algorithm's end, the PT and the OT contain the following information.

```
     PT                          OT

T1     committed        O1   restored    vm address
T2     prepared         O2   restored    vm address
```

Scenario 2--mutex objects

Suppose the situation depicted in Figure 3-8 exists at a participant's log after a crash. O_1 and O_2 are mutex objects. Action T_1 committed, and action T_2 prepared successfully, but aborted later.

Unlike built-in atomic objects, a mutex object has only one version, namely, the *current* version. On recovery the current version of a mutex object is the last data entry written in the log by an action that prepared successfully (the *prepared* outcome entry for that action is in the log), regardless of whether said action later aborted or committed.

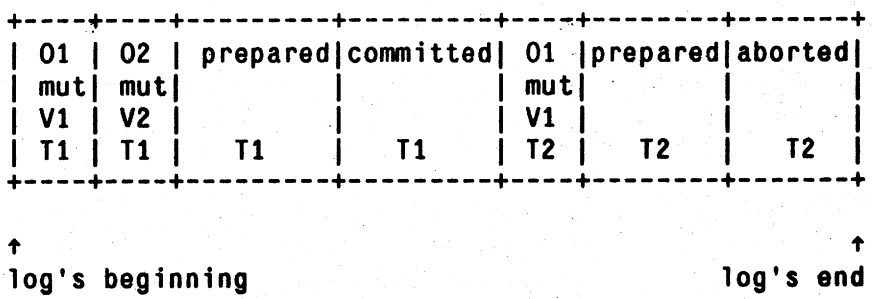


Figure 3-8: Log of mutex objects following a crash

According to the semantics for mutex, what matters is that the action *prepared*. We explain how recovery is done for the recoverable objects in the log depicted in the figure.

The recovery system reads the log backwards starting with the last outcome entry:

1. For outcome entry <aborted,T_2>, the recovery system enters the action id into the PT and sets the action state to *aborted*.

2. For outcome entry <prepared,T_2>, the recovery system checks the PT and notices that action T_2 has aborted; the entry is ignored.

3. For data entry <O_1,mutex,V_1,T_2>, the recovery system sees that action T_2 has aborted. Because V_1 is a mutex object version, it is the current version of mutex for the object, even though the action that modified it aborted. The object uid is not already present in the OT. The recovery system copies V_1 into volatile memory and remembers the address of the object. It enters the uid O_1 into the OT, sets the object state to *restored*, and sets the vm address to the object address.

4. For outcome entry <committed,T_1>, the recovery system enters the action id T_1 into the PT and sets the associated action state to *committed*.

5. For outcome entry <prepared,T_1>, the recovery system checks the PT. The action state associated with the action id T_1 is already *committed*, so this entry is ignored.

6. For data entry <O_2,mutex,V_2,T_1>, the recovery system checks the PT and knows that action T_1 has committed. It looks up O_2 in the OT; it is not present in the OT so the recovery system copies the object version V_2 into volatile memory and remembers the object address. The object uid is entered into the OT with object state *restored* and the vm address of the object.

7. For the <O_1,mutex,V_1,T_1> data entry, the recovery system checks the PT and sees that action T_1 has committed. The object uid is already in the OT, and although object O_1 is a mutex object it is not copied because a later version has been copied.

8. The recovery system makes a final pass over volatile memory, replacing any uid

references with their appropriate volatile memory references. The stable counter is reset to O_2.

9. The recovery system creates an empty AS, traverses the stable state in volatile memory from the stable variables, and fills the new AS with the uids of accessible objects. In this case, the new AS contains uids O_1 and O_2, assuming that O_1 and O_2 were accessible.

10. The PT and OT tables are returned to the Argus system in order to resume the participant.

At algorithm's end, the PT and OT contain the following information.

```
      PT                                OT

T1      committed        O1   restored    vm address
T2      aborted          O2   restored    vm address
```

Scenario 3--newly accessible recoverable objects

The situation depicted in Figure 3-9 exists at a guardian's log, after a crash. This situation would arise as a result of the scenario described in Figure 3-5 occurring before the crash.

```
+--+--+--------+---------+--+--+--+--------+--+--------+-------+---------+
|bc|bc|prepared|committed|O1|bc|O3|prepared|O2|prepared|aborted|committed|
|O1|O2|        |         |at|O3|at|        |at|        |       |         |
|V1|V2|        |         |V1|V3|V3|        |V2|        |       |         |
|  |  |  T1    |   T1    |T2|  |T2|   T2   |T3|  T3    |  T2   |    T3   |
+--+--+--------+---------+--+--+--+--------+--+--------+-------+---------+

↑                                                                       ↑
log's beginning                                                   log's end
```

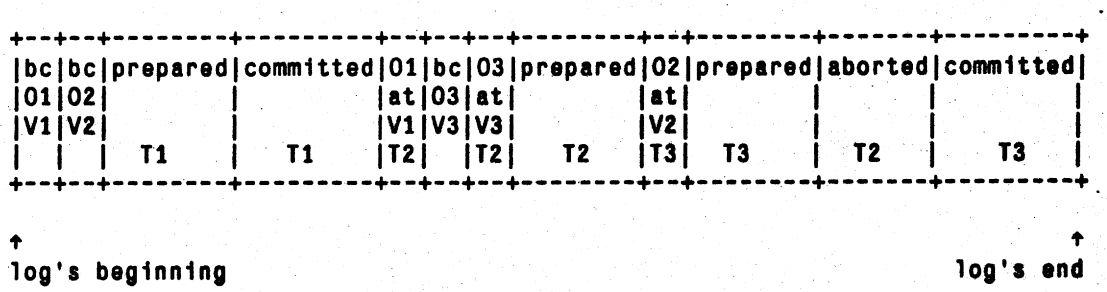


Figure 3-9: Log following a crash

The recovery system reads the log backwards starting with the last outcome entry.

1. For outcome entry <committed,T_3> the recovery system enters the action id T_3 into the PT and sets the associated action state to *committed*.

2. For outcome entry <aborted,T_2> the recovery system enters the action id T_2 into the PT and sets the associated action state to *aborted*.

3. For outcome entry <prepared,T_3> the recovery system checks the PT and sees that the action has already committed, and ignores the entry.

4. For data entry <O_2,at,V_2,T_3> the recovery system checks the PT for T_3 and finds that the action is committed. Since O_2 is not in the OT, it copies V_2 into volatile

memory as the current version and remembers the volatile memory address. The object uid O_2 is entered into the OT, the object state is set to *restored*, and the vm address is set to this object address.

5. For outcome entry <prepared,T_2> the recovery system ignores this entry because the action state associated with the action id is *aborted*.

6. For data entry <O_3,atomic,V_3,T_2>, the recovery system checks the PT and notices that T_2 has aborted. Since the object is atomic, the recovery system ignores the entry.

7. For outcome entry <bc,O_3,V_3>, the recovery system knows that this is a base version. It checks the OT, and since it is not present, the recovery system copies V_3 to volatile memory as the base version for an atomic object and remembers the address of the copied object in volatile memory. The recovery system enters the object uid O_3 into the OT, sets the object state to *restored*, and sets the vm address to the object address.

8. For data entry <O_1,atomic,V_1,T_2>, the recovery system checks the PT and sees that the action id has action state *aborted*, which means that the action aborted. Since the object is atomic, the entry is ignored.

9. For outcome entry <committed,T_1>, the recovery system enters the action id T_1 into PT with associated action state *committed*.

10. For outcome entry <prepared,T_1>, the recovery system consults the PT and sees that the action id has action state *committed*. The entry is ignored.

11. For outcome entry <bc,O_2,V_2>, the recovery system knows that V_2 is a base version. It checks the OT and finds O_2 with object state *restored*, so the entry is ignored because the latest version has already been copied.

12. For outcome entry <bc,O_1,V_1>, the recovery system knows that V_1 is a base version. It consults the OT, does not find O_1 there, and copies V_1 to volatile memory as a base version, remembering the address in volatile memory. It enters O_1 into the OT, sets the object state to *restored*, and sets the vm address to the object address.

13. The recovery system makes a final pass over volatile memory, replacing any remaining uid references with their appropriate volatile memory references. The stable counter is reset to O_3.

14. The recovery system creates an empty AS, traverses the stable state in volatile memory from the stable variables, and fills the new AS with the uids of accessible objects. In this case, the new AS contains uids O_1, O_2, and O_3, since the objects are accessible.

15. The PT and OT are returned to the Argus system in order to resume the participant.

At algorithm's end, the PT and OT contain the following information.

```
        PT                          OT

T1      committed        O1  restored   vm address
T2      aborted          O2  restored   vm address
T3      committed        O3  restored   vm address
```

Notice that the stable state of the guardian in volatile memory following recovery will look exactly like the situation that existed before the crash in Step 8 of Figure 3-5, which is what we wanted.

Scenario 4

Suppose the situation depicted in Figure 3-10 exists at a guardian's log, after a crash.

```
+--+--+--+--------+---------+--+--------+----------+---------+----+
|bc|O1|bc|prepared|committed|O2|prepared|committing|committed|done|
|O1|at|O2|        |         |at|        |          |         |    |
|V1|V1|V2|        |         |V2|        | P1,P2,P3 |         |    |
|  |T1|  |   T1   |   T1    |T2|   T2   |    T2    |   T2    | T2 |
+--+--+--+--------+---------+--+--------+----------+---------+----+
↑                                                                 ↑
log's beginning                                          log's end
```

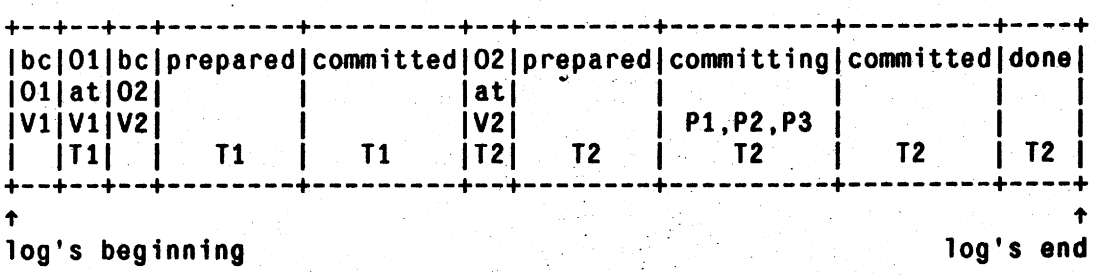


Figure 3-10: Coordinator's log following a crash

In this scenario we show the entries that are written to the log for the coordinator of an action during two-phase commit.

To recover the objects from the guardian's log in Figure 3-10, we need to extend the algorithm to include coordinators. Let us add a third table, which stores information about coordinator states. Thus,

CT: action id → coordinator action state

where coordinator action state = {committing, done}. *committing* contains a list of the guardian identifiers that were involved in the action.

Notice that in the guardian's log a particular ordering of outcome entries holds true if the top-level action committed successfully: *prepared*, *committing*, *committed*, and *done*. Why? When each participant has prepared, it forces the *prepared* outcome entry to its log. The coordinator, upon hearing that everyone has prepared, forces the *committing* entry to

indicate that the coordinator is now in the second phase of two-phase commit. The coordinator instructs the participants to commit, and each does so by forcing the *committed* entry in the log. When all participants have informed the coordinator that they have committed, the coordinator forces *done* to the log; the top-level action has completed.

The system reads the log backwards starting with the last outcome entry.

1. For outcome entry <done,T_2> the recovery system knows that action T_2 has completed two-phase commit. It enters the action id into the CT and sets the coordinator action state to *done*.

2. For outcome entry <committed,T_2>, the recovery system enters the action id, T_2, into the PT and sets the participant action state to *committed*.

3. For outcome entry <committing,<P_1,P_2,P_3>,T_2>, the recovery system checks the CT. The associated action state is *done*, which means the action already completed, so this entry need not be considered any further. If, however, the action was not *done* then the system would enter the action id into the CT, set the action state to *committing* and include the list of guardian identifiers of guardians that were participating in two-phase commit.

4. For outcome entry <prepared,T_2>, the recovery system checks the PT and notices that the associated action state is *committed*. The recovery system ignores the entry (if the action has already committed, then it must have prepared).

5. For data entry <O_2,atomic,V_2,T_2>, the system looks up the action id T_2 in the PT and finds that the action had committed. It then checks the OT for object uid O_2, finds that it is not there, copies V_2 into volatile memory as the current version, and remembers the volatile memory address. The object uid is entered into the OT, the object state is set to *restored*, and the vm address is set to the object address.

6. For outcome entry <committed,T_1>, the recovery system inserts T_1 into the PT and sets the action state to *committed*.

7. For outcome entry <prepared,T_1>, the recovery system checks the PT and notices that the associated action state is *committed*. This entry is ignored.

8. For outcome entry <bc,O_2,V_2>, the recovery system knows that V_2 is a base version. It checks the OT, finds the object uid present with object state *restored*, and ignores the entry.

9. For data entry <O_1,atomic,V_1,T_1>, the recovery system consults the PT for T_1 and notices that the action is *committed*. Since O_1 is not in the OT, the recovery system copies the object version V_1 into volatile memory as the current version for an atomic object and remembers the object address. It then enters it into the OT, sets the object state to *restored*, and sets the vm address to the object address.

10. For outcome entry $\langle bc, O_1, V_1 \rangle$, the recovery system knows that the object version is a base version. It checks the OT for the object uid. The object state is *restored*, so the recovery system knows that the object version had already been copied, and ignores the entry.

11. The recovery system makes a final pass over volatile memory, replacing any remaining uid references with their appropriate volatile memory references. The stable counter is reset to O_2.

12. The recovery system creates an empty AS, traverses the stable state in volatile memory from the stable variables, and fills the new AS with the uids of accessible objects. In this case, the new AS contains uids O_1 and O_2, assuming that O_1 and O_2 were accessible.

13. The CT, PT, and OT tables are returned to the Argus system in order to resume the respective tasks of the participants and coordinators.

At algorithm's end, the participant's action table (PT), the coordinator's action table (CT), and the object table (OT) contain the following information.

```
      PT                    CT                    OT

T1    committed      T2     done      O1  restored  vm address
T2    committed                       O2  restored  vm address
```

Since the table contains no action identifier whose state is *committing* then no coordinator needs to be restarted.

3.4.3 Turning uids into pointers

Given some object version in a data entry, a *base_committed* entry, or a *prepared_data* entry, we have not exactly said how uids in the version are changed to volatile memory addresses. We address this issue in this section.

Suppose we have an object version in a data entry that is to be restored to volatile memory. The volatile memory format of the object is constructed for this object version but the uid is replaced with the volatile memory address of a special object containing the uid, assuming that the object's address in volatile memory is not already known. (Since uids are not necessarily the same size as addresses, we require a different format.) Now when the recovery system makes a final pass over volatile memory it follows pointers and checks to see what they are pointing at. If a pointer points to a special object containing the uid, the recovery system checks the OT for the uid and fetches the reference to the real object in volatile memory. This reference replaces the reference to the special uid object.

3.4.4 The General Recovery Algorithm

The following is the general algorithm for recovering objects from a guardian's log.

1. Create three tables:

   a. An object table (OT) that maps object uids to object states--*prepared* or *restored*--and volatile memory addresses of objects. Each entry is of the form <uid, object state, object vm address>, where the uid is the lookup key.

   b. A coordinator action table (CT) that maps action ids to coordinator action states: *committing* and *done*. Each entry is of the form <*aid*, coordinator action state>, where the *aid* is the lookup key. In addition, the *committing* state has a list of guardian ids of the participants involved in an action.

   c. A participant action table (PT) that maps action ids to participant action states: *prepared*, *committed*, and *aborted*. Each entry is of the form <*aid*, participant action state>, where the *aid* is the lookup key.

2. Read the log *backwards*, starting with the last outcome entry in the log. $\forall$ log entry, process it in the following way.

   a. *prepared* outcome entry. If *aid* $\in$ PT then ignore the entry. If *aid* $\notin$ PT then insert <*aid*, *prepared* state> in the PT.

   b. *committed* outcome entry. Insert <*aid*, *committed* state> in the PT.

   c. *aborted* outcome entry. Insert <*aid*, *aborted* state> in the PT.

   d. *base_committed* outcome entry. If uid $\in$ OT then check the object state: if *prepared* then copy the object version to volatile memory as the base version, and set the object state to *restored*. If uid $\notin$ OT then copy the object version to volatile memory as the base version if the object is atomic, and insert <uid,*restored* state, object vm address> into the OT.

   e. *prepared_data* outcome entry. Check the PT:

      i. If *aid* $\in$ PT and has participant action state *aborted* then the entry is ignored. If *aid* $\in$ PT and has state *committed*, then check the OT. If uid $\in$ OT then consider the object state: if *prepared* then copy the object version to volatile memory as the base version of an atomic object and set the object state to *restored*; if *restored* then ignore the entry. If uid $\notin$ OT then copy the object version to volatile memory as the base version and insert <uid, *restored* state, object vm address> into the OT.

      ii. If *aid* $\notin$ PT, then the action must have prepared (the real *prepared* outcome entry appears earlier in the log), and so <*aid*, *prepared* state> is entered into the PT. The object version is copied to volatile memory as the current version of an atomic object and the *aid* is

granted a write lock. <uid, *prepared* state, object vm address> is inserted in the OT.

f. *committing* outcome entry. If *aid* $\in$ CT then ignore the entry. If *aid* $\notin$ CT then insert <*aid*, *committing* state(gids)> into the CT, where gids are the guardian identifiers of the participants.

g. *done* outcome entry. Insert <*aid*, *done* state> in the CT.

h. *Data* entry. The recovery system checks the PT for the *aid* of this entry.

i. If *aid* $\in$ PT and the associated participant action state is *committed* then check the OT for the object uid. If uid $\in$ OT then consider the object state. If *prepared* state then copy the object version to volatile memory as the base version if the object is atomic (if mutex, then ignore entry) and reset the object state to *restored*; if *restored* state then ignore the entry because the object has already been copied. If uid $\notin$ OT then copy the object version to volatile memory as the base version if the object is atomic or as the current version if the object is mutex. Insert <uid, *restored*, object vm address> into the OT.

ii. If *aid* $\in$ PT and the associated participant action state is *prepared* then check the OT for the object uid. If uid $\notin$ OT then copy the object version to volatile memory: if atomic object, then copy as the current version, grant *aid* a write lock, and insert <uid, *prepared*,object vm address> into OT; if mutex object, then copy the version and insert <uid, *restored*,object vm address> into OT. If uid $\in$ OT and object state is *restored*, then ignore the entry because the object has already been copied.

iii. If *aid* $\in$ PT and the participant action state is *aborted* then consider the object type. If the object is atomic, then the recovery system ignores this entry. If the object is mutex, then check the OT for the object uid. If uid $\in$ OT then ignore the entry because the mutex object has already been copied. If uid $\notin$ OT then copy the object version to volatile memory as the current version and insert <uid, *restored*,object vm address> into the OT.

3. The recovery system makes a final pass over volatile memory, replacing object uid references with volatile memory references. The stable counter (used to generate uids) is reset to the largest uid stored in the OT. Now, the recoverable objects have been reconstructed in volatile memory, and the guardian's stable state has been restored.

4. The recovery system creates an empty accessibility set and traverses the stable state once more, this time constructing a new set with object uids so that it now represents the actual stable state.

5. These three tables are returned to the Argus system in order to resume the tasks

of the participants and coordinators.

Chapter Four

Hybrid Log -- Writing and Recovery Algorithms

The previous chapter set forth the basic ideas, in the context of the simple log, concerning the format of log entries, the algorithm for writing recoverable objects to the log, and the algorithm for recovering these objects from the log. In this chapter we first reiterate the advantages and disadvantages of both the pure log scheme and the shadowed objects scheme, as covered in Chapter 1, and argue in favor of a *hybrid log*, which combines the virtues of both schemes. Next, we present the new format for data and outcome entries for the hybrid log and, rather than explain again the algorithm for writing recoverable objects to the hybrid log, we note just the differences from Chapter 3's exposition. Then we present the new algorithm that recovers objects from the hybrid log, and again, only the differences from Chapter 3 are emphasized. Finally, we explain the notion of *early prepare* and the extent to which it affects the writing and recovery algorithms for the hybrid log.

4.1 Simple log versus Hybrid log

In Chapter 1 we introduced two possible approaches for organizing stable storage, namely, the pure log scheme and the shadowed objects scheme. We noted that these two schemes actually represented the two ends of a spectrum of stable storage organization and pointed out that other schemes indeed existed between these extremes. Let us summarize the advantages and disadvantages of these two schemes:

1. Log ⇒ fast writing, but slow recovery

2. Shadowing ⇒ slow writing, but fast recovery

Naturally, we would like an organization that permits both fast writing and fast recovery, but that is probably an unattainable ideal in practice. Instead, we have identified a scheme that falls between the two ends of the spectrum and thus is a kind of hybrid of the pure log scheme and the shadowed objects scheme. What are the advantages of this hybrid

log scheme over the other two schemes? First, it has the virtue of the pure log in that writing is fairly fast (append-only memory makes writing fast). Second, it has the virtue of shadowed objects in that the table is incrementally written into a log entry for easy access to objects in the log, which makes recovery fairly fast, though slower than the shadowed objects scheme in its unadulterated form.

Let us examine this hybrid log in more detail.

4.2 Writing objects to the log

The hybrid log scheme retains the structure of the log and looks very much like the simple log we dealt with in the last chapter, but with a basic difference--the format of the log entries themselves. Where does the map from the shadowed objects scheme fit in? Rather than being maintained as a single entity, the map is distributed over the log entries, in particular, the *prepared* outcome entries; in other words, the map is implicit in all the *prepared* outcome entries put together.

Figure 4-1 shows the new format of log entries for the hybrid log. There are three differences between the new format and and old format of entries for the simple log. First and perhaps the most crucial difference is that the *prepared* outcome entry now contains information in addition to the action identifier of the preparing action. A list of <uid,log address> pairs where the uid is the object's unique identifier and the log address is the address of the data entry containing the object version. One pair is created for each recoverable object that was written to the log as a data entry for the action. Notice that this is a portion of the map from the shadowed objects scheme. Second, data entries no longer need the action ids and object uids since the *prepared* outcome entries contain that information. Third, each outcome entry--*prepared*, *committed*, *aborted*, *committing*, *done*, *base_committed*, and *prepared_data*--has another field, the log pointer field that contains the log address of an outcome entry. This field is used to link each outcome entry in the log to the previous outcome entry, forming a backward chain of outcome entries: the head of the chain is the last outcome entry and the end of the chain is the very first outcome entry.

Suppose we have the hybrid log, depicted in Figure 4-2 after an action has prepared. During the prepare phase of two-phase commit, the recovery system writes data entries to

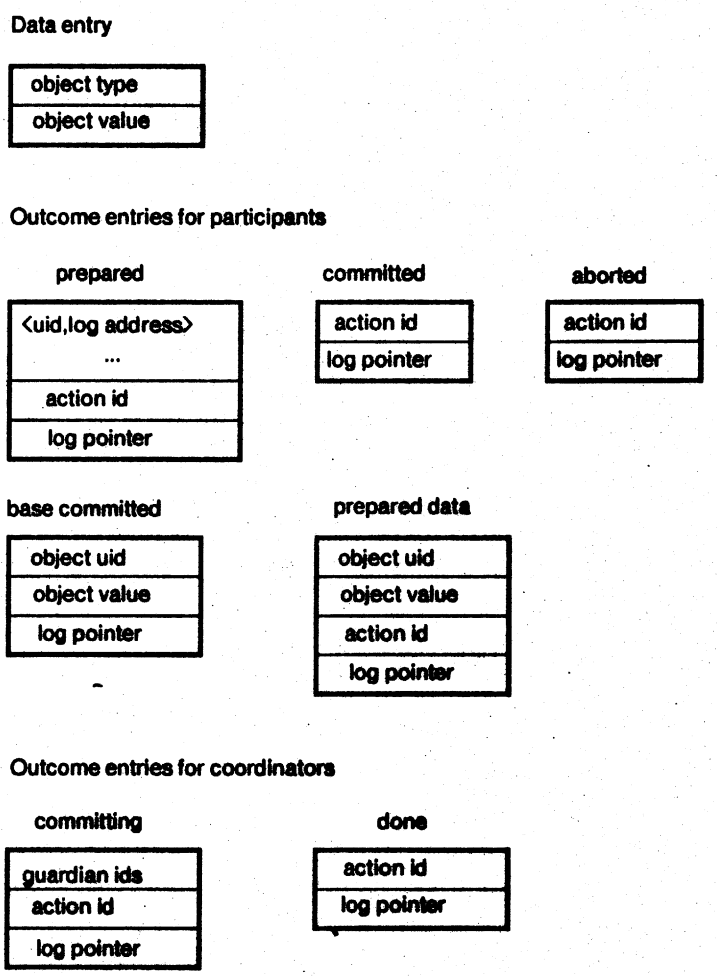


Figure 4-1: New format of log entries

the participant's log for some action and internally keeps track of the object uids and the log addresses of the data entries. When it is finished, it creates a *prepared* outcome entry consisting of the list of <uid, log address> pairs and the log address of the previous outcome entry and forces the entry to the log. Notice that the recovery system must keep track of this information for every preparing action. The only other difference is that each of the other outcome entries is linked via the log pointer field to the previous outcome entry before it is forced to the log.

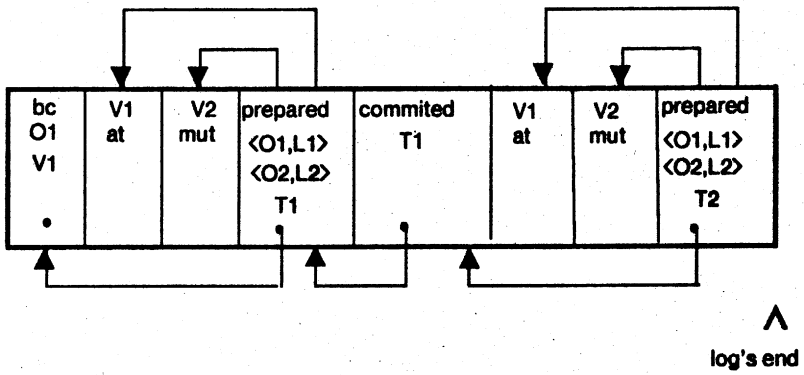


Figure 4-2: Log after the prepare phase

4.3 Recovering objects from the log

In this section we present a sketch of the general recovery algorithm. One log scenario demonstrates the manner in which the new recovery algorithm recovers objects from the log. We then give a detailed explanation of the differences between this recovery algorithm and the simple log's recovery algorithm.

4.3.1 Sketch of the General Algorithm

1. Create three tables: (1) an object table (OT) that maps object uids to both object states (*prepared* or *restored*) and object locations in volatile memory, (2) a coordinator action table (CT) that maps action ids to coordinator action states (*committing* and *done*, where *committing* also has a list of guardian ids of guardians involved in the action), and (3) a participant action table (PT) that maps action ids to participant action states (*prepared*, *committed*, and *aborted*).

2. Read the log backwards, starting with the last outcome entry in the log. For every *outcome* entry on the backward chain of outcome entries, process it in the following way:

    a. If the outcome entry is *aborted*, *committed*, *committing*, or *done* then fill the three tables with appropriate information (action ids and action states like *prepared*).

    b. If the outcome entry is a *prepared* entry, then for each <uid, log address> pair in the entry check the OT and determine whether or not to copy the object version into volatile memory; if it needs to copy the object version it follows the log address pointer to the data entry itself.

c. If the outcome entry is *base_committed* or *prepared_data* then, if the object is not already present in the OT, update the OT and copy the object version to volatile memory for an atomic object.

3. Make a final pass over volatile memory, replacing object uid references with volatile memory references. Once this pass is completed, all recoverable objects have been reconstructed, and the guardian's stable state has been restored.

4. Create a new, empty AS, and traverse the stable state, filling the AS with the uids of those objects that are actually accessible.

5. Return these three tables to the Argus system in order to resume the tasks of the participants and coordinators.

4.3.2 Log Scenario and Recovery

Suppose the situation depicted in Figure 4-2 exists at a guardian's log after a crash. O_1 is a built-in atomic object and O_2 is a mutex object. The stable state of this guardian is restored in the following way.

The recovery system reads the log backwards starting with the last outcome entry.

1. For outcome entry $\langle prepared, \langle\langle O_1, L_1\rangle, \langle O_2, L_2\rangle\rangle, T_2\rangle$ the recovery system checks the PT for the action id, enters it since it is not there, and sets the participant action state to *prepared*.

   a. For the $\langle O_1, L_1\rangle$ pair the recovery system checks the OT for the uid, finds it is not there, follows the L_1 pointer to the data entry, and copies the object version (V_1) to volatile memory as the current version for an atomic object; T_2 is granted a write lock and the recovery system remembers the object address. The recovery system enters the uid into the OT, sets the object state to *prepared*, and sets the vm address to the object address.

   b. For the $\langle O_2, L_2\rangle$ pair the recovery system checks the OT, follows the log address L_2 to the data entry and copies the object version (V_2) to volatile memory as the current version for a mutex object, and remembers the object address. It also enters the uid into the OT, sets the object state to *restored*, and sets the vm address to the object address.

   The recovery system follows the log pointer to the previous outcome entry.

2. For outcome entry $\langle committed, T_1\rangle$ the recovery system enters the action id T_1 into PT and sets the participant action state to *committed*. It follows the log pointer to the previous outcome entry.

3. For outcome entry $\langle prepared, \langle\langle O_1, L_1\rangle, \langle O_2, L_2\rangle\rangle, T_1\rangle$ the recovery system looks up the action id T_1 in the PT and finds that the action had already committed.

a. For the $\langle O_1, L_1 \rangle$ pair the recovery system looks up O_1 in the OT and finds it present with object state *prepared*. Since the action also committed, this is the latest committed version and the recovery system follows the log address L_1 to the data entry and copies the object version V_1 to volatile memory as the base version of O_1. It also resets the object state to *restored*.

b. For the $\langle O_2, L_2 \rangle$ the recovery system looks up O_2 in the OT and finds it present with object state *restored*. The recovery system does nothing since the object version has already been copied to volatile memory.

The recovery system follows the outcome entry log pointer to the previous outcome entry.

4. For $\langle bc, O_1, V_1, nil \rangle$, the recovery system does not bother checking the PT but consults the OT. The object uid already appears in the OT with object state *restored*, so the system ignores the log entry and goes on. The address of the log pointer field for this log entry is nil, so we are finished.

5. The recovery system makes a pass over volatile memory, translating object uid references to their volatile memory references. Once this pass is completed, all recoverable objects have been reconstructed, and the guardian's stable state has been restored. The stable counter is reset to O_2.

6. The recovery system creates a new, empty AS, and traverses the stable state, filling the AS with the uids of those objects that are actually accessible. In this case, the AS consists of uids O_1 and O_2, assuming that the objects are accessible.

7. The CT, PT, and OT are returned to the Argus system, so that the respective tasks of the participants and coordinators can be resumed.

At algorithm's end, the object table (OT), the participant's action table (PT), and the coordinator's action table (CT) contain the following information:

```
        OT                          PT                  CT

O1   restored  vm address     T1    committed
O2   restored  vm address     T2    prepared
```

4.3.3 The General Recovery Algorithm

In this section we point out how Chapter 3's general recovery algorithm is changed to accommodate the hybrid log scheme.

First, instead of processing every log entry (both data and outcome) as the log is read backwards, the recovery system processes every *outcome* entry, each of which is linked to the previous outcome entry, and when it has finished with one outcome entry it follows the

log pointer in the entry to the previous outcome entry. This continues until there are no more outcome entries, at which time recovery is complete. Thus, the main loop is as follows:

```
∀ outcome entry on backward chain do
    process it
    copy appropriate object versions to volatile memory
    end
```

Second, the *prepared* outcome entry, as noted before, contains more information. In particular, it contains the object uids and log pointers to the corresponding data entries; these objects were either modified by the preparing action or made newly accessible. The recovery system processes each <uid,log address> pair in virtually the same way it did for data entries except that it need not follow the log pointer to the data entry unless it has to copy the object version.

Otherwise, the recovery algorithm for this hybrid log scheme is the same as in the simple log scheme.

4.4 Early prepare

In Chapter 3 we made a simplifying assumption: objects modified by an action are written to the log *only* during the prepare phase of two-phase commit. This assumption made it possible to explain the recovery algorithm through simple scenarios without burdening the reader with certain complications. Having laid the groundwork in the previous chapter and modified it slightly for the hybrid log scheme covered in this chapter, we now relax our assumption. The objects modified by a committing top-level action are written to the log sometime before the *prepare* message is actually sent, which makes preparing potentially faster.

The idea is to take advantage of free time in the guardian, if any. Rather than waiting for a top-level action to prepare and then writing out the data entries to the log all at once, it might be better to write out changes early, that is, in anticipation of the prepare of the top-level action. In this way, if the action eventually commits just the *prepared* and *committed* outcome entries are written; if it aborts then extra work has been done, but that is not a problem because we assume that aborts are not as frequent as commits.

The method of writing data to the log in anticipation of a commit is called *early prepare*. One result of early prepare is that data entries written on behalf of different actions

are interleaved in the log; this introduces some changes in the recovery system operations, the writing algorithm, and the recovery algorithm. Let us consider these changes.

First, we add one recovery system operation, namely, *write_entry(aid, MOS)*. This operation allows the Argus system to "early prepare" the objects in the MOS for action *aid* at a guardian. The operation also returns a set of objects that were not written to a guardian's log because they were inaccessible; we will explain in a moment why this new set is returned. The *prepare(aid,MOS)* operation remains essentially the same and the MOS contains objects that had not already been early prepared and that will be prepared.

Second, for each action the Argus system keeps track of which objects were early prepared and which were not. The new operation mentioned above takes a set of objects modified by an action (MOS), which are to be early prepared, and writes the accessible objects to the log in the manner prescribed in Chapter 3. In addition, the operation returns a set of objects (MOS') that were not early prepared because they were inaccessible. Since they might later turn out to be accessible this MOS' becomes the new MOS, and the next time the operation will be called with this MOS, which may also contain more objects to be early prepared. Finally, when the *prepare* message is received, any objects modified by the preparing action but not already early prepared will be written to the log in the usual way.

Now we explain the added complication in the recovery algorithm through an example. See Figure 4-3. The problem here is that the recovery algorithm will not copy the latest object version for mutex object O_1. How did this situation arise?

1. Action T_1 seized the mutex lock on object O_1, modified the object, and released the mutex lock. The recovery system wrote out the data entry for the object version to the log as part of early prepare.

2. Another action, T_2, seized the mutex lock on object O_1 and modified it. The recovery system wrote out the object version to the log as a data entry.

3. Two more data entries were written on behalf of action T_2.

4. The participant received the *prepare* message for T_2 from its coordinator, and the recovery system created a *prepared* outcome entry with the proper information and forced it to the log.

5. A data entry for object O_4 was written to the log on behalf of action T_1.

6. The participant received a *prepare* message for T_1 from its coordinator. The recovery system created the *prepared* outcome entry with the proper

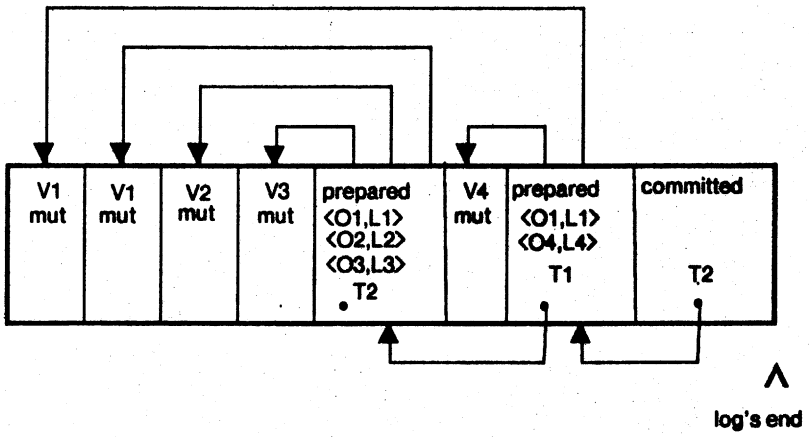


Figure 4-3: Hybrid log after T1 prepares and T2 commits

information and forced it to the log for action T_1.

7. The participant received a *commit* message for T_1 from its coordinator. The recovery system created the *committed* outcome entry with the proper information and forced it to the log.

8. The Argus system crashed.

On recovery we see that the earlier version, rather than the latest version, of O_1 gets copied to volatile memory, which is wrong. To solve this problem, we need to keep some extra information in the OT for mutex objects, namely, the log address of the "latest" data entry for that object that had been copied from the log. When we encounter another data entry for that object, we compare its log address with the one stored in the OT. If the new address is less than the old one, then the recovery system ignores the entry. If the new address is greater, then the recovery system copies the object version in the data entry to volatile memory and updates the OT with this data entry's log address. Also, the vm address field is updated with the new address of the object version.

Chapter Five

Hybrid Log -- Housekeeping Algorithms

We have seen that the log is a repository of all the recoverable objects that were ever part of the guardian's stable state during the guardian's lifetime. As time goes on and actions commit and abort, the log will grow and will eventually become quite large. A large log has certain implications for recovery.

Under the simple log scheme, the recovery system would be forced to read *every* log entry upon recovery from a crash in order to reconstruct the stable state, including data entries whose objects had been already been copied to volatile memory. If the log is as large as we have postulated then recovery might be unacceptably slow. The same is true, but to a lesser extent, of the hybrid log scheme presented in the last chapter; here the recovery system would read every *outcome* entry (rather than every log entry) in the log and need not examine every data entry. Even so, if the log is large, then recovery might be slower than we would like.

If we place a bound on how much of the log the recovery system would have to look at in order to restore the stable state, this would speed up recovery as a consequence since the recovery system would have a shorter log to deal with. In this chapter we discuss techniques that can reduce the size of the log.

In particular, recovery can be speeded up if we could produce a *checkpoint* that represents the stable state of a guardian. The checkpoint's size would be roughly proportional to the number of recoverable objects making up the stable state. The recovery system would record a checkpoint of the stable state by building a new stable state from the recoverable objects that were written by recently committed actions. The key words here are *recently committed*, because by concerning ourselves with just the recently committed recoverable objects and constructing a new stable state from them, we effectively throw away not only all the objects modified by aborted actions but also all previous versions of objects modified by committed actions.

As the checkpoint proceeds the new stable state is written to a new log as a series of distinct data entries (one for each recoverable object), which are linked together by a special log entry called *committed_ss*. This new log replaces the old log when the checkpoint has completed. During recovery the recovery system reads the log backwards, and if it encounters this entry, it finishes restoring the guardian's stable state given the information in that entry.

When does the recovery system checkpoint a guardian's stable state? It should take place at any time as determined by the Argus system and as frequently as needed.

The recovery system can checkpoint the stable state of a guardian in one of two ways: it can either build a new stable state from the log entries themselves ("compacting the log") or copy the stable state in volatile memory ("taking a snapshot"). We discuss these methods in the next two sections and then we compare the two methods.

5.1 Compacting the log

The basic idea behind compacting the log is to rummage through the whole log, discarding old objects that are no longer useful and retaining the necessary ones, thus creating a new log that is smaller, but that still reflects accurately the guardian's current stable state. The recovery system starts a new process running in parallel with ordinary recovery system operations. This process reads the old log just as in recovery from a crash and writes log entries to a new log, while the recovery system continues to write to the old log. When the process has reached the beginning of the old log it then copies to the new log all outcome entries and their associated data entries written to the old log since compaction began. At some point all writing to the old log is frozen while the last outcome entry in the old log is written to the new log and, in one atomic step, the new log supplants the old log. This log compaction technique is a kind of garbage-collection because most of the information in the old log can be discarded, leaving behind a new log that contains just the information about recoverable objects written by recently committed actions as well as some extra information about prepared actions.

5.1.1 The Compaction Algorithm

Suppose at some point the Argus system determines that a certain amount of log activity has taken place and decides that a guardian's log should be compacted. After some log entry is written to the log, the recovery system sets a volatile variable to point to the end of that log entry. We call this point the *housekeeping* marker. Notice that the housekeeping marker divides the log into two parts: the part to be compacted and the part to be copied over to the new log when compaction is completed.

Compaction happens in two stages. In the first stage, we pick out the data entries for all recoverable objects modified by recently committed actions as well as the data entries for prepared actions, and we write new log entries to a new log. While this copying is going on, the recovery system is still writing entries to the old log beyond the housekeeping marker. In the second stage, these entries are copied to the new log. At some point we freeze further writing to the old log and the remaining entries are copied to the new log, and then the new log supplants the old one. The recovery system resumes writing entries to the new log. We now discuss these stages in more detail.

The first stage of compacting the log is similar to recovery after a crash, except that where the objects would have been created in volatile memory, data entries are written to the new log instead. The recovery system starts a new compaction process and sets the housekeeping marker. The process creates an empty list, the *outcome entries list* (OEL); as each outcome entry is written to the old log after the housekeeping marker by the ordinary recovery system operations its log address is recorded in this list, and so the order of outcome entries in this list corresponds to the order in the old log. Why we need this list will become clearer when we discuss the second stage. The process creates another list, the *committed stable state list* or CSSL for short. This list will contain the <object uid, new data entry log address> pairs, which correspond to those data entries in the new log whose object versions were written by committed actions; they constitute the committed stable state of the guardian. The process also creates the three tables that are ordinarily used on recovery. The participant action table (PT) and the coordinator action table (CT) have exactly the same meaning as before; the object table (OT) is also the same except that it need not record the volatile memory address. The compaction process creates these three

tables, and in addition, creates a *new*, empty log. It begins reading the old log backwards from the housekeeping marker, starting with the last outcome entry. Let us consider each outcome entry.

1. *committed*, *aborted*, and *done* outcome entries. Simply update the PT.

2. *committing* outcome entry. If the outcome is not yet known, then create a new *committing* outcome entry containing the same information, link it to the previous outcome entry in the new log, and write it to the new log.

3. *base_committed* outcome entry. Check the OT. If the uid is present with state *restored* then go on because the object has already been copied. Otherwise, enter the uid in the OT with state *restored*, create a new data entry containing a copy of the object version and the object uid, link it with the previous outcome entry in the new log, write it to the new log, and insert <uid, data entry log address> in the CSSL.

4. *prepared_data* outcome entry. Check the PT for the action outcome.

    a. If the action has prepared or its outcome is unknown, then create a new *prepared_data* outcome entry containing the the object uid, the object version, and the action id, link it to the previous outcome entry in the new log, and write it to the new log. Enter uid in OT with state *prepared*.

    b. If the action has aborted, then ignore the entry.

    c. If the action has committed, then check the OT. If the uid is present in the OT with state *restored*, then ignore the entry. If the uid is present with state *prepared*, reset state to *restored*, create a new data entry containing the object version and the object type atomic, write it to the new log, and insert the <uid, data entry log address> into the CSSL. If the uid is not present, enter the uid with state *restored* into the OT, create a new data entry containing the object type atomic and the object version, write it to the new log and insert <uid, log address> in the CSSL.

5. *prepared* outcome entry. Check the PT for the action outcome.

    a. If the action has aborted, then for every <uid, log address> pair in the outcome entry do the following. Read the data entry. If the object type is atomic, then move on to the next pair; if the object type is mutex, check the OT and ask whether this mutex is the most *recent* version (compare the log address stored in the OT with the old log address of the data entry under consideration). If so, then create a new data entry containing the object version and the object type mutex, write it to the new log, insert the <uid, new data entry log address> in the CSSL, and update the OT with the old data entry log address. If not, then ignore the entry.

    b. If the action has committed, then for every <uid, log address> pair in the list do the following. Read the data entry. If the object type is atomic, then check the OT. If the uid is in the OT with state *restored*, then go on

to the next pair in the list; if the uid is not in the OT, then insert the uid with state *restored* into the OT, or if the uid is present with state *prepared* then reset the state to *restored*; in either case, create a new data entry containing the object version and the object type, write it to the new log, and insert the <uid, new data entry log address> in the CSSL. If the object type is mutex, then proceed in the same fashion as for aborted actions above.

c. If the action outcome is not known, then create an empty *prepare list* that will contain the <uid, new data entry log address> pairs. For every <uid, log address> pair in the outcome entry do the following. The old *prepared* outcome entry will be copied over to the new log as a new but slightly altered *prepared* outcome entry. Read the data entry. If the object type is atomic, insert uid into the OT with state *prepared*, write a new data entry to the new log containing the object version and the object type, and insert a <uid,new data entry log address> in the prepare list, not the CSSL. If the object type is mutex, then check the OT to see if it is the latest version. If it is the latest version then insert <uid,data entry log address> in the OT, write the entry to the new log, and record the <uid, new data entry log address> in the CSSL; otherwise, ignore it and go on to the next pair. When the old prepare list is exhausted, if the new prepare list is not empty, then create a new *prepared* outcome entry containing this new prepare list, link it with the previous outcome entry in the new log, and write it to the new log.

When the process has reached the beginning of the log, it creates a *committed_ss* log entry containing the CSSL, links it with the previous outcome entry, and writes it to the new log. Notice that this entry is like a combined prepare and commit for some special action whose name does not matter.

In the second stage, the compaction process considers the OEL; this list represents all the outcome entries that had been written to the old log *after* the housekeeping marker, and these outcome entries must be copied over to the new log. Starting with the first entry in the OEL, the housekeeping process, in general, follows the log pointer to the outcome entry, creates a new outcome entry with a copy of the same information, links it with the previous outcome entry in the new log, and writes it to the new log. This is all that need be done for all outcome entries except *prepared*.

For the *prepared* outcome entry the process does the following. For each <uid, log address> pair in the prepare list the process reads the data entry. If the object is atomic then it creates a new data entry containing the object version and object type atomic, writes it to the new log, and places the <uid, data entry log address> in the new prepare list. If the object is mutex, then the process compares the old log address of the data entry with the log

address stored in the OT associated with the uid: if the data entry log address is less than the OT log address then the data entry is ignored; otherwise, a new data entry is created containing the object version and object type mutex and written to the new log, the <uid,data entry log address> is placed in the new prepare list, and the OT is updated with the old log address.

When the process has finished processing the old prepare list then it creates a new *prepared* outcome entry with this new prepare list (together with the action id), links it with the previous outcome entry in the new log, and writes it to the new log. When the process catches up with the recovery system that is writing entries to the old log and updating the OEL, the recovery system stops writing entries to the old log while the compaction process copies the last outcome entry in the old log to the new one. Finally, in one atomic step, the compaction process replaces the old log with the new log, thus discarding the old log. The recovery system resumes writing log entries to the new log. The compaction process terminates, its job done.

The above algorithm will not copy data entries from the old log for which the *prepared* outcome entry has not yet been written. These data entries are not lost forever because the recovery system knows which actions have not yet prepared and restarts the writing of the data entries for those actions to the new log when compaction is over.

5.1.2 The New Recovery Algorithm

The new recovery algorithm is virtually identical to the recovery algorithm presented in the last chapter, with one exception, which has to do with the the *committed_ss* log entry. The recovery system reads the log backwards and goes about restoring the guardian's stable state in the usual fashion. When it encounters the *committed_ss* log entry it treats it as a commit and prepare of an anonymous action. It cycles through the CSSL, checking the OT in the usual way, and if necessary copies the object version to volatile memory. When the recovery system has reached the first outcome entry all the objects that should be in volatile memory have been reconstructed and volatile memory now contains the stable state of the guardian. The three tables are returned to the Argus system in order to resume guardian activity.

5.2 Taking a snapshot of the stable state

Rather than reading the guardian's log backwards as in recovery, the recovery system might just copy the guardian's stable state that resides in volatile memory while actions execute. We call this alternate technique the *stable state snapshot*.

Whenever the Argus system has determined that enough log activity has taken place it tells the recovery system to begin housekeeping, that is, to reorganize the log in a way that makes recovery efficient after a crash. In response, the recovery system starts a snapshot process.

Like log compaction, taking a snapshot also happens in two stages. In the first stage, the recovery system copies the actual stable state into a new log. In the second stage, it copies log entries to the new log from the housekeeping marker forward in the old log.

We will consider just the first stage in the snapshot algorithm because the second stage is essentially the same as for the log compaction algorithm. There is another table that the recovery system maintains itself, updates as actions execute, and uses while the snapshot takes place: a *mutex table* (MT). This table maps object uids for mutex objects to the log addresses of the corresponding data entries in the old log. This table must be maintained during all recovery system activity; unlike the OEL (Outcome Entries List), its use is not restricted to just the duration of housekeeping. The reason we need this table is that without it we cannot be sure whether the volatile version of a mutex object encountered during the snapshot is the same as the latest prepared version that appears in the log. In the case of mutex objects, the real information needed for recovery is recorded in the log, not in volatile memory. We must copy to the new log only the latest versions of those mutex objects that were modified by actions that prepared. When an action prepares and an object version for some modified mutex object is written to the old log, if the object uid is not in the MT then it is entered with the data entry log address; otherwise, the log address in the MT is changed to be the new data entry log address.

The snapshot process creates an empty CSSL (committed stable state list), an OEL (outcome entries list), a new log, a new AS (accessibility set), a new, empty MT, and sets the housekeeping marker in the old log. The recovery system adds log addresses of outcome entries to the OEL as the entries are written to the old log. Meanwhile, the snapshot process

begins traversing the graph of recoverable objects from each of the guardian's stable variables and adding uids to the new AS. Suppose that it encounters a recoverable object during its traversal. There are two cases depending on whether the object is atomic or mutex.

First, suppose the object is atomic. The object is either read-locked or write-locked by some action, or not locked at all. If it is read-locked or not locked then the snapshot process can safely copy the base version of the object, create a new data entry, write it to the new log, and place the <object uid, data entry log address> pair in the CSSL. If the object is write-locked by an action, the process checks the PAT (Prepared Actions Table): (1) if the action id is not in the table, then just the base version is copied, a new data entry is created and written to the new log, and the <object uid, data entry log address> pair is placed in the CSSL; (2) if the action id is in the PAT, then the action has prepared. The process creates a new data entry containing the copied *base version*, which is written to the new log, and the <uid,data entry log address> is inserted in the CSSL. The process also creates a new *prepared_data* outcome entry that contains the object uid, copied *current version*, and action id, links it with the previous outcome entry, and appends it to the new log.

Second, suppose the object is mutex. The process looks up the object uid in the old MT. If the object uid exists then it follows the pointer associated with the uid to the corresponding data entry in the old log and creates a new data entry containing the copied object version, which is appended to the new log and whose <object uid, new data entry log address> pair is placed in the CSSL. Also, the log address of this new entry is stored in the new MT. If the object uid does not exist, then the mutex version is *not* copied (but will be copied if needed in stage two of the snapshot method). In this case we have a newly accessible object, and the action that made it accessible is preparing. When that action finishes preparing, the right state for this object will be written to either the old log (in which case it will be copied to the new log in stage two) or it will be written directly to the new log. Therefore, we need not worry about writing this object now.

The second stage of the snapshot algorithm is the same as in the second stage of the compaction algorithm, except that we must update the new MT as needed. At the end the new AS is intersected with the old AS to yield a new set, which is the new accessibility set.

Also, the old MT is replaced by the new MT.

In this second stage, some data might be copied unnecessarily. In particular, some data entries for objects where the snapshot has already copied the correct information will be written anyway. Extra copying can only happen for objects modified or made newly accessible by actions that prepared while the snapshot was being written. The amount of extra copying is proportional to the amount of processing that occurred against the old log while the snapshot was being made. We assume that the amount of such processing is limited, so the extra copying is not significant.

The new recovery algorithm is almost the same as the one covered in the section on log compaction. The MT must be reconstructed on recovery after a crash. During recovery, the log address of the each data entry containing the latest prepared version of a mutex object is entered in the MT.

5.3 Summary

In this chapter we discussed two techniques for reorganizing the log to make recovery from crashes more efficient. Collectively, we referred to these techniques as *housekeeping* and the end result was a new, smaller log. Under either technique the recovery system divided the log into two parts by setting a housekeeping marker: the portion of the log before the marker would be reorganized and copied to a new log, and the portion of the log following the marker, indicating additional guardian activity, would then be copied to the new log. The difference lies in how it treats the portion of the log before the marker. In log compaction, the recovery system actually reads the log itself backwards from the marker, just as if it were performing recovery, and constructs a new stable state from the entries in the log. In taking a snapshot, the recovery system traverses the recoverable objects (accessible from the guardian's stable variables) that constituted the stable state actually residing in volatile memory, and constructs a new stable state from those objects.

In general, we think that the snapshot technique of reorganizing the log is better, for the following reason. The advantage of this method is that it takes an amount of time roughly proportional to the number of accessible recoverable objects; the compaction method would take much longer since it must process all outcome entries as well as all

accessible objects. The disadvantage of the snapshot is the space required for the MT and the time used in keeping the MT up to date in volatile memory. The time required to update the MT should be insignificant since the MT can be organized as a hash table; therefore, only the space consumed by the MT is significant. We expect that it will be worthwhile to trade this space for the time saved.

Chapter Six

Conclusions

In this thesis we investigated the mechanism needed to provide data resiliency for on-line, long-lived data in a distributed computer system. Recall that data resiliency meant the ability of data to survive hardware failures such as crashes of nodes or storage devices, with high probability. There were two aspects to the problem of providing this data resiliency: implementing stable storage devices and organizing the use of stable storage. *Assuming* that stable storage devices existed and were at our disposal, we addressed the question of *how* stable storage could be organized to make recovery from crashes efficient.

Having looked at two different schemes for organizing stable storage that are commonly found in the literature, namely, the *log* and *shadowing* schemes, we invented yet another organization called the *hybrid log* that combined the advantages of both schemes. This organization retained the characteristics of the pure log to keep writing fairly fast and incorporated the advantage of shadowed objects to speed up recovery. Our decision was influenced to a large degree by an assumption: crashes were infrequent and therefore normal processing should be fast at the possible expense of a slow recovery. In short, there was a tradeoff between the speed in writing to stable storage and the speed in recovering from a crash, and we favored writing over recovering. However, we attempted to strike a balance between writing and recovering.

Once we chose the organization of stable storage that met our requirements, we developed algorithms for writing objects to the hybrid log, recovering objects from the hybrid log, and reorganizing the hybrid log. Reorganizing the log, which we call housekeeping, involves changing the way the log looks so that recovering from a crash is much faster than it would be if we just let the log continue to grow. We investigated two housekeeping schemes--log compaction and stable state snapshot--and concluded that the snapshot technique was strictly better than the log compaction technique.

We have implemented a prototype to increase our confidence that the algorithms

behave as they should. More work remains to be done, however. At one extreme is the verification of the algorithms. We need to state precisely what the correctness properties are for the algorithms and then verify that the algorithms preserve those properties. For atomic objects the property is that the state of each object after a crash is exactly what is obtained from running all actions that committed at a guardian in their serial order. For mutex objects, however, the property is not so easy to state because of the semantics of Argus that requires recovery of all mutex versions written for a prepared action.

At the other extreme is a real implementation of the recovery system and its algorithms. The system must then be run in support of "realistic" applications and its performance measured. In this way we will be able to evaluate the efficiency of the algorithms, and we will be able to validate or disprove the assumptions on which the recovery system is based.

Finally, the recovery system is based on an abstraction of stable storage, the stable log. This abstraction must be implemented using real storage devices in a way that provides the needed reliability.

References

[Arens 81] Arens, Gail C. *Recovery of the Swallow Repository*. Technical Report MIT/LCS/TR-252, M.I.T. Laboratory for Computer Science, January, 1981. Master's thesis.

[Astrahan 76] Astrahan, Morton M., et al. "System R: A Relational Approach to Database Management". *ACM Transactions on Database Systems* 1(2):97-137, June, 1976.

[Bjork 75] Bjork, L. A. "Generalised Audit Trail Requirements and Concepts for Data Base Applications". *IBM Systems Journal* 14(3):229-245, 1975.

[Davies 73] Davies, Charles T. "Recovery Semantics for a DB/DC System". In *Proceedings of the 1973 ACM National Conference*, pages 136-141. 1973.

[Davies 78] Davies, Charles T. "Data Processing Spheres of Control". *IBM Systems Journal* 17(2):179-198, February, 1978.

[Eswaran 76] Eswaran, Kapal P., Gray, James N., Lorie, Raymond A., and Traiger, Irving L. "The Notion of Consistency and Predicate Locks in a Database System". *Communications of the ACM* 19(11):624-633, November, 1976.

[Gray 78] Gray, James N. "Notes on Database Operating Systems". In Goos and Hartmanis (editors), *Lecture Notes in Computer Science 60*, pages 393-481. Springer-Verlag, Berlin, 1978.

[Gray 81] Gray, James N., et al. "The Recovery Manager of the System R Database Manager". *Computing Surveys* 13(2):223-242, June, 1981.

[Lampson 79] Lampson, Butler W. and Sturgis, Howard E. "Crash Recovery in a Distributed Data Storage System". 1979.Xerox Palo Alto Research Center,Palo Alto, California (April, 1979). Unpublished paper.

[Liskov 82] Liskov, Barbara H. and Scheifler, Robert W. "Guardians and Actions: Linguistic Support for Robust Distributed Programs". In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 7-19. January, 1982. Revised version to appear in *ACM Transactions on Programming Languages and Systems*.

[Moss 81] Moss, J. Eliot B. *Nested Transactions: An Approach to Reliable Distributed Computing*. Technical Report MIT/LCS/TR-260, M.I.T. Laboratory for Computer Science, June, 1981. Ph.D. thesis.

[Raible 83] Raible, Eric. "A Log-based Interface to Stable Storage for the Argus Language". 1983.Bachelor's Thesis, M.I.T. Department of Electrical Engineering and Computer Science. May, 1983.

[Reed 81] Reed, David P. and Svobodova, Liba. "Swallow: A Distributed Data Storage System for a Local Network". In West, A. and Janson, P. (editors), *Local Networks for Computer Communication*, pages 355-373. North Holland Publishing Company, 1981.

[Svobodova 80] Svobodova, Liba. *Management of Object Histories in the Swallow Repository*. Technical Report MIT/LCS/TR-243, M.I.T. Laboratory for Computer Science, July, 1980.

[Weihl 82] Weihl, William E. and Liskov, Barbara H. "Specification and Implementation of Resilient Atomic Data Types". 1982.Available as Computation Structures Group Memo 223, M.I.T. Laboratory for Computer Science, December, 1982. To appear in *ACM SIGPLAN '83*: a Symposium on Programming Language Issues in Software Systems.

CS-TR Scanning Project
Document Control Form

Date : 4 / 4 / 96

Report # LCS-TR-308

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 76 (82-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
(A) IMAGE MAP!	(1-76) UNI'ED TITLE PAGE, 2-76 (77-82) SCANCONTROL, DOD(2), TRGT'S(2)
(B) HALF INCH FROM THE BOTTOM OF EACH PAGE IS A LINE [redacted] FROM XEROXING.	

Scanning Agent Signoff:
Date Received: 4/4/96 Date Scanned: 4/22/96 Date Returned: 4/22/96

Scanning Agent Signature: Michael W. Cook

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: MIT/LCS/TR-308	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Reliable Object Storage to Support Atomic Actions	5. TYPE OF REPORT & PERIOD COVERED: Interim Research
	6. PERFORMING ORG. REPORT NUMBER: MIT/LCS
7. AUTHOR(s): Brian Masao Oki	8. CONTRACT OR GRANT NUMBER(s): DARPA/DOD N00014-83-K0125; NSF 82-03486MCS
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: DARPA/DOD/IPTO, 1400 Wilson Boulevard, Arlington, VA 22209	12. REPORT DATE: October 1983 13. NUMBER OF PAGES: 76
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research/Dept of Navy, Information Systems Program, Arlington, VA 22217	15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; distribution is unlimited.

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

Unlimited

18. SUPPLEMENTARY NOTES

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

Atomic actions, atomic objects, distributed systems, logs, recovery, shadowing, stable storage, transactions.

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

To preserve the consistency of on-line, long-lived, distributed data in the presence of concurrency and in the event of hardware failures, it is necessary to ensure atomicity and data resiliency in applications. The programming language Argus is designed to support such applications. This thesis investigates the mechanism needed to support the notion of data resiliency present in Argus. Data resiliency means that the probability is very high that the crash of a node or storage device in a distributed system does not cause the loss of

...continued

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Continued ...

vital data. Data resiliency requires the use of stable storage devices, memory devices that survive failure to a high probability. This thesis is not concerned with how to implement stable storage devices, but rather with how to organize the use of stable storage. The thesis presents a new organization of stable storage called the hybrid log that provides fast writing of information to stable storage and reasonably fast recovery of information from stable storage. In the context of this scheme, various algorithms are developed for writing objects to the log, recovering objects from the log, and housekeeping the log.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned
Date: 04/19/1996
M.I.T. Libraries
Document Services