# Reliable Object Storage to Support Atomic Actions

by

*Brian Masao Oki*

May 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

# Table of Contents

# Table of Figures

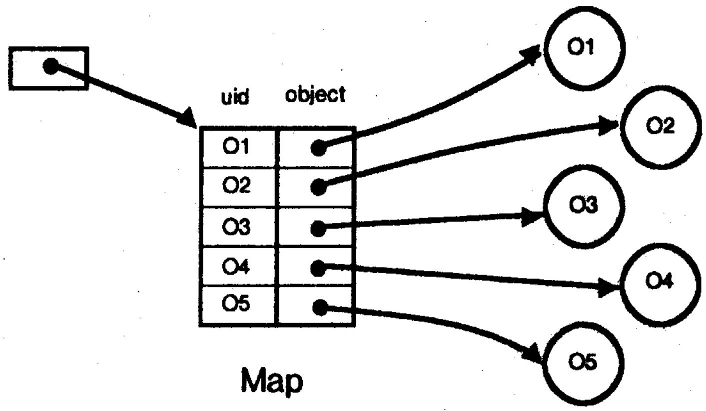



**Figure 1-1:** Shadowed objects

### 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 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.

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

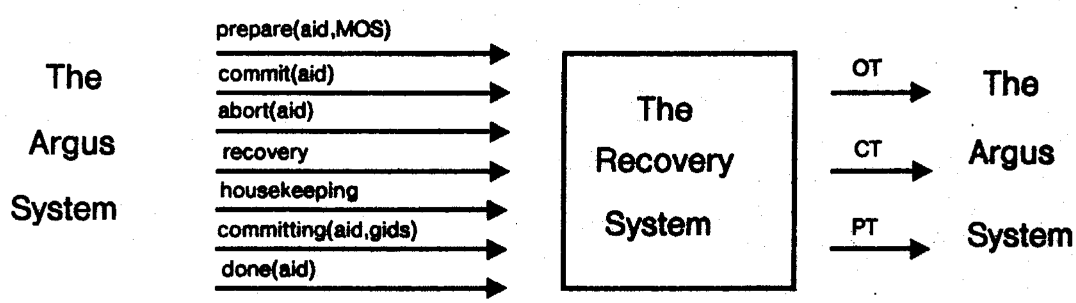


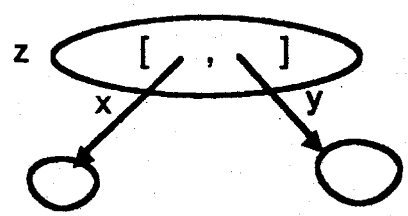

**Figure 2-1:** The Recovery System

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.

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.

(*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.

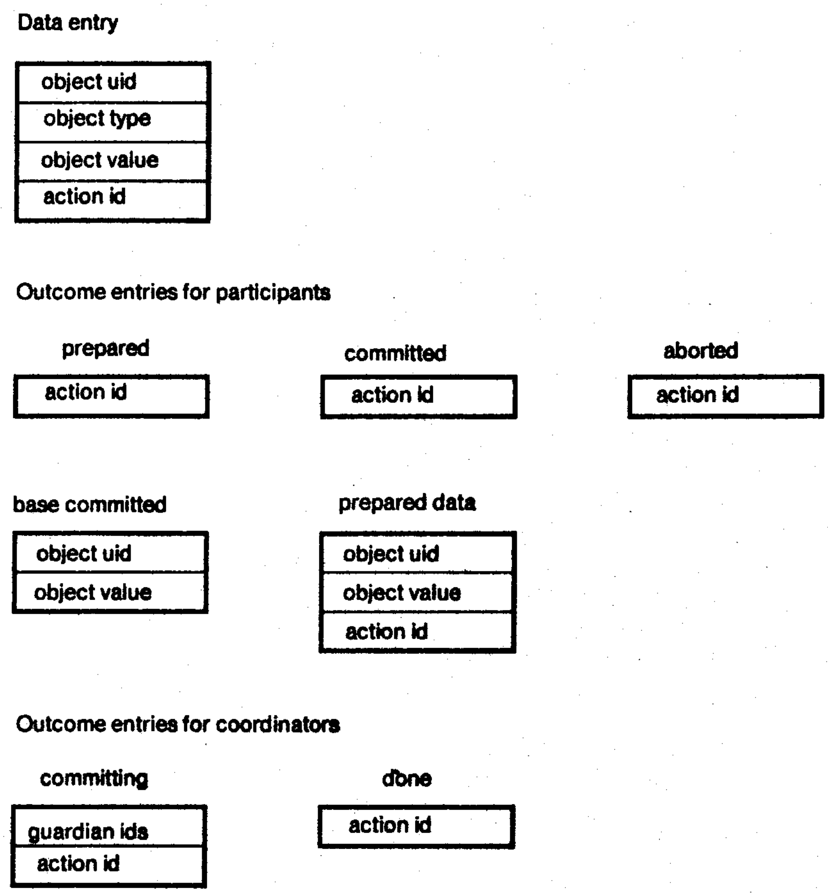



**Figure 3-1:** Data entries and Outcome entries

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 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 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.

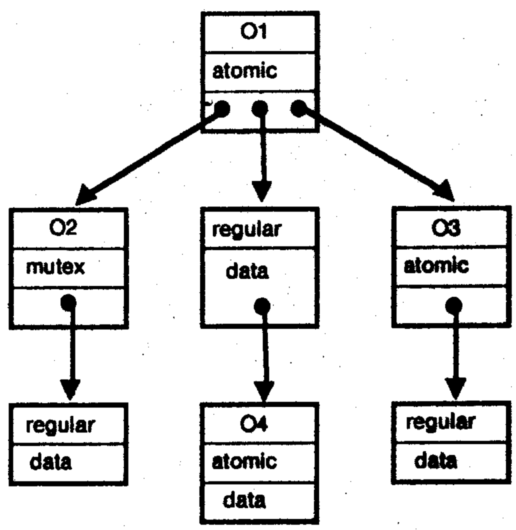



**Figure 3-3:** Objects in volatile memory

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.

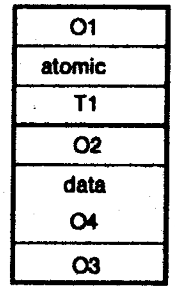



**Figure 3-4:** Flattened Object

### 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.

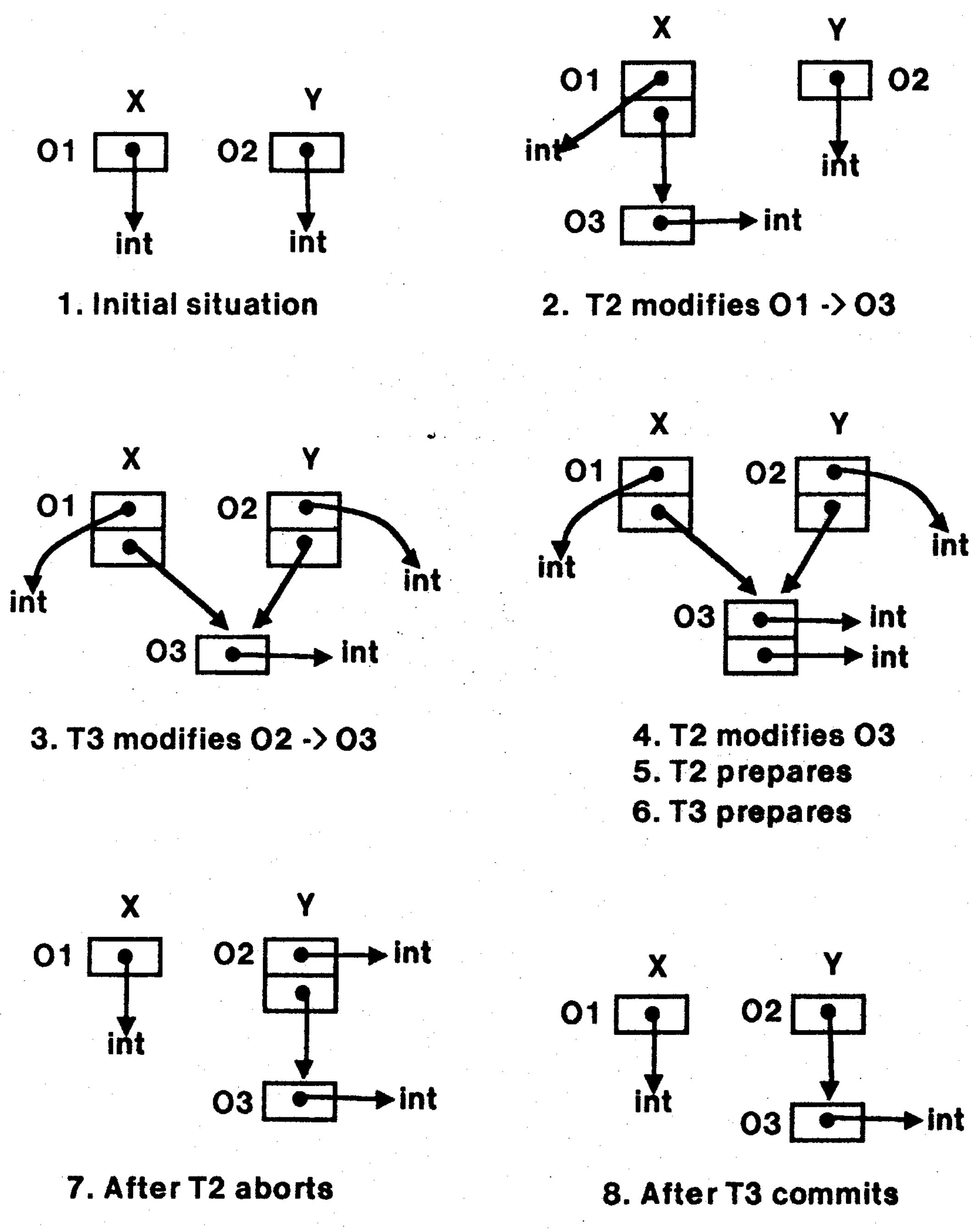
Notice that this concern with accessible objects is really an optimization because we could simply write out *all* the recoverable objects at a guardian without regard for accessibility or inaccessibility; if some inaccessible object were written out to stable storage it would not matter since it was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

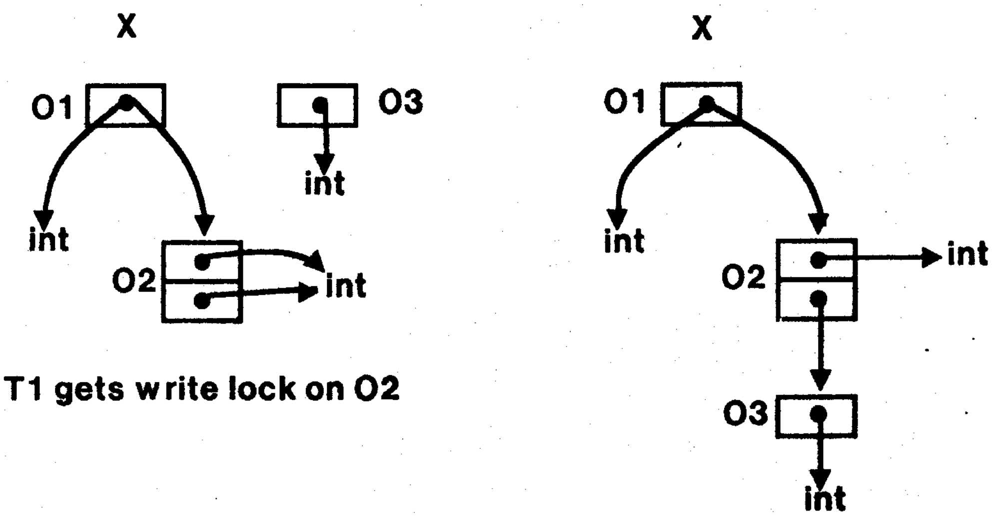
Figure 3-5: Newly Accessible Objects Example



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 $O_1$, $O_2$, $O_3$.




**a. T1 gets write lock on O2**

**b. T1 modifies O2 to point to O3**

**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

### 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.

```
+----+----+----+--------+---------+----+--------+
| bc | bc | 02 |prepared|committed| 01 |prepared|
| 01 | 02 | at |        |         | at |        |
| V1 | V2 | V2 |        |         | V1 |        |
|    |    | T1 |   T1   |   T1    | T2 |   T2   |
+----+----+----+--------+---------+----+--------+

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

**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*.

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

```
         PT                         OT

T1       committed          01   restored    vm address
T2       aborted            02   restored    vm address
T3       commited           03   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|01|bc|prepared|committed|02|prepared|committing|committed|done|
|01|at|02|        |         |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 of the participants and coordinators.

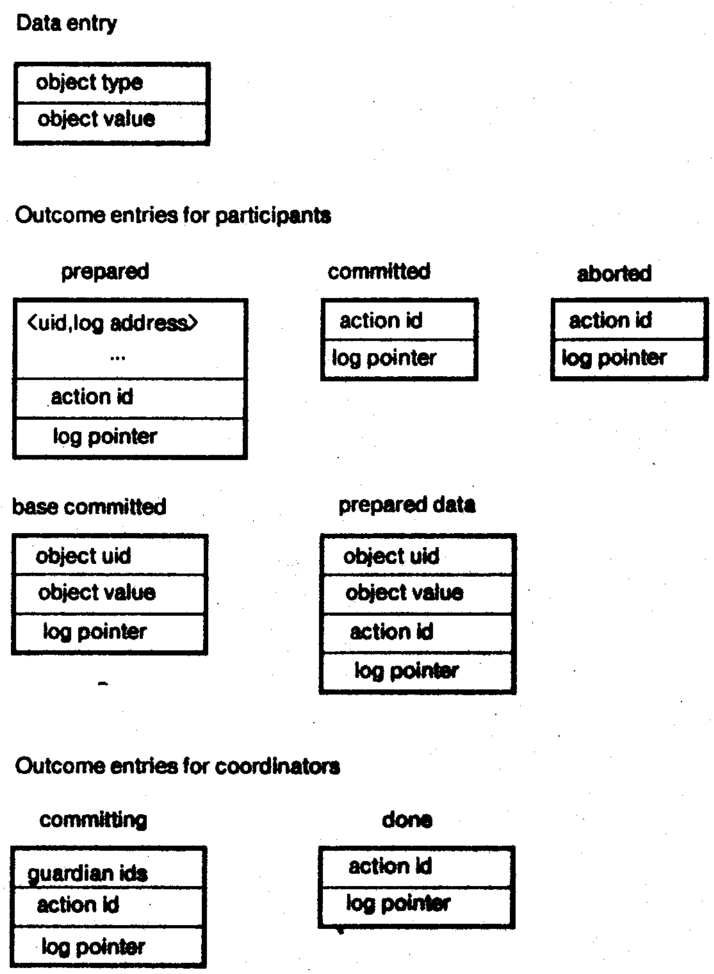



**Figure 4-1:** New format of log entries

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.

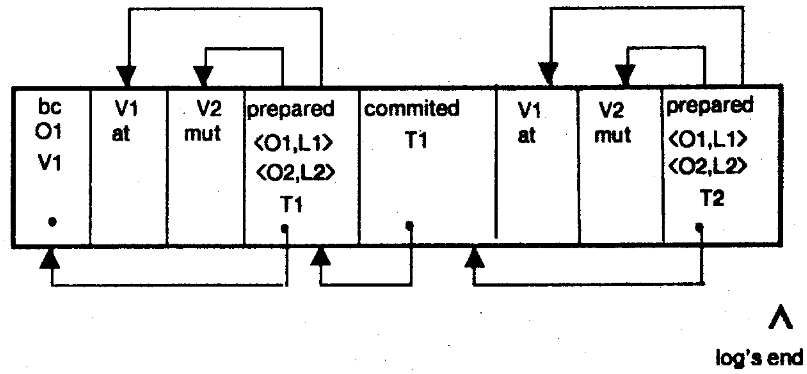



**Figure 4-2:** Log after the prepare phase

## 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.

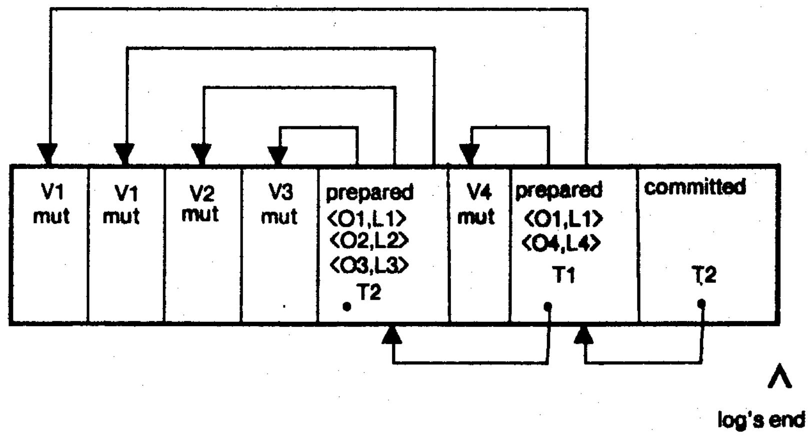



**Figure 4-3:** Hybrid log after T1 prepares and T2 commits

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.

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.

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.

[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.