Reliable Object Storage to Support Atomic Actions

Brian M. Oki, Barbara H. Liskov, and Robert W. Scheifler

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

Abstract

Maintaining consistency of on-line, long-lived, distributed data in the presence of hardware failures is a necessity for many applications. The Argus programming language and system, currently under development at M.I.T., provides users with linguistic constructs to implement such applications. Argus permits users to identify certain data objects as being resilient to failures, and the set of such resilient objects can vary dynamically as programs run. When resilient objects are modified, they are automatically copied by the Argus implementation to stable storage, storage that with very high probability does not lose information. The resilient objects are therefore guaranteed, with very high probability, to survive both media failures and node crashes.

This paper presents a method for implementing resilient objects, using a log-based mechanism to organize the information on stable storage. Of particular interest is the handling of a dynamic, user-controlled set of resilient objects, and the use of early prepare to minimize delays in user activities.

1. Introduction

In banking systems, airline reservation systems, office automation systems, and other databases, the manipulation and preservation of long-lived, on-line, distributed data is of primary importance. The Argus programming language and system [12], currently under development at M.I.T., is designed to support such applications. A fundamental requirement in such systems is making data resilient to hardware failures, so that the crash of a node or storage device will not result in the loss of vital information. This paper discusses support for data resiliency in Argus.

In Argus, data consistency in the presence of concurrency is achieved by making activities *atomic*. Atomic activities are referred to as *actions* or transactions [4, 5, 6]. An action is *indivisible* and *total*. Indivisibility means that the execution of one action never appears to overlap the execution of any other action. Totality 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*). While an action is running, the changes it makes to data objects are kept in volatile storage. If the action aborts, the changes are simply discarded. If the action commits, however, the changes become permanent. Our method of providing data resiliency is to write such changes to stable storage.

Stable storage provides memory with a high probability of surviving node and media failures [11]. A stable storage device might provide block read and write operations just like a conventional disk device; the write operation, however, is *atomic*, meaning the data is either written completely or not written at all, even if there is a failure during the write. This atomicity 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. Lampson and Sturgis [10] call this kind of stable storage *atomic stable*

storage, and describe one method for implementing it. In this paper, we ignore the details of implementing stable storage itself and focus instead on how stable storage can be organized to allow a distributed system to recover from failures efficiently.

In most existing work on databases [2], file repositories [8], and object repositories [16, 17, 1], users must make explicit calls on the system to create, modify, or delete resilient data: the data is under explicit system control. Argus is unique in its integration of resilient data into the fabric of the programming language. In Argus, the programmer divides data between stable state and volatile state; only stable state survives node crashes. The stable state is specified as a fixed collection of *root* objects. Objects, however, can refer to other objects, and inter-object references can be changed dynamically under program control. The stable state is actually the set of all objects *reachable* from the root objects. Thus, objects enter and leave the stable state implicitly rather than explicitly. The job of the recovery system is to write data objects to stable storage as needed, to restore the data objects after a crash, and to reorganize stable storage to make crash recovery more efficient.

The next section presents background information on the Argus programming language and its model of computation, including atomic actions and the two-phase commit protocol. Section 3 presents a log-based organization of stable storage. Subsequent sections present the algorithms for writing objects to the log and for recovering objects from the log, and a snapshot technique for speeding up crash recovery. We conclude with a discussion of current status and future plans.

2. Background

This section lays the groundwork for understanding the remaining sections. We first explain the basic concepts underlying Argus, particularly atomic actions and atomic objects, and then discuss how Argus implements atomictiy.

2.1. The Programming Language Argus

Argus [12] gives programmers the ability to write distributed programs that run on a network of heterogeneous computers. Each node in the network is an independent computer consisting of one or more processors with local memory and devices; nodes communicate with each other only by sending messages over the network.

A distributed Argus 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 obtain access to the called guardian's resources. In addition to processes executing handler calls, a guardian can have processes to perform background tasks.

Guardians are the logical nodes of the distributed system. Each resides at a single physical node, although a node may support several guardians. When a guardian's node crashes, the volatile state of the guardian, as well as all processes, are lost; only the stable state survives. The roots of the stable state are defined by a fixed number of statically declared variables, called the *stable variables*. The set of all objects accessible from these variables (via all chains of inter-object references) constitutes the stable state; these objects are called *stable objects*. When the guardian's node recovers from a crash, the Argus system re-creates the guardian with its stable objects as recorded on stable storage. A user process is then started in the guardian to reinitialize the volatile state. Once this task is completed, the guardian's background processes are restarted, and the guardian starts accepting handler calls.

Atomic actions are the primary method of performing distributed computations in Argus. The effect of an action is all-or-nothing; that is, it either completes successfully (commits) and changes the state permanently, or fails completely (aborts) and restores the state that existed before the action was executed. An action starts at one guardian and can spread to other guardians by means of handler calls. (Actually, handler calls are run as subactions of the calling action as is discussed further in section 2.2 below.) When the action completes, it either commits at all guardians, and the changes it made to each guardian's stable state are reflected in stable storage appropriately, or it aborts at all guardians.

Atomicity is achieved with *atomic objects*. Argus provides several built-in types of atomic objects, such as atomic_arrays and atomic_records. These are similar to ordinary objects except that they use locks and versions to provide totality and indivisibility for actions using the objects. There are two kinds of locks: read locks and write locks. To use an object, an action must invoke one of the object's operations. The operation acquires the lock in the appropriate mode and the action holds the lock until it completes (commits or aborts). When a write lock is first obtained for an action, a *version* of the object is made in volatile memory, and the action operates on this version, called the *current* version. The old version, called the *base*

version, is also retained. If the action commits, the current version becomes the base version and the old base version is discarded. If the action aborts, the current version is discarded.

Argus also supports user-defined atomic objects (see [18]), which present an external interface that supports atomicity, but can offer significant concurrency as well. User-defined atomic objects do not have a major impact on the algorithms described in this paper; to simplify the presentation, we consider only built-in atomic objects. Complete algorithms are presented in [15].

2.2. Overview of Transaction Processing in Argus

As actions execute, modifications to objects are made on volatile versions at several guardians. As we mentioned above, each object is contained in exactly one guardian. Each guardian keeps track of all uses of its own objects; such information is never sent to another guardian. In particular, for each action that visits a guardian, the guardian records every atomic object the action reads and every one it modifies. Information about these modified objects is maintained in the Modified Objects Set (MOS); a guardian maintains a separate MOS for every action that has visited it and that has not yet committed or aborted.

When an action commits, the system must ensure that it either commits everywhere or aborts everywhere, and that its effects are made permanent by writing the modified atomic objects to stable storage. Since the guardians themselves know which objects were modified by the action, the system just needs to communicate with the guardians that the action visited, using the standard *two-phase commit protocol* [7]. Since the algorithms presented later in this paper are tightly coupled with this protocol, we explain it briefly here. Although many optimizations of the standard protocol are possible, we avoid describing any of them for the sake of simplicity. The protocol works even if crashes occur while it executes; we assume that no nodes crash forever and eventually any two nodes can communicate [14]. Following standard terminology, we call the guardian where the action originates the *coordinator*, and the various guardians visited via handler calls the *participants*.

1. **Coordinator's Preparing phase.** In the preparing phase, the coordinator sends a *prepare* message to each participant (including itself) saying "prepare for action A to commit," where A is the action identifier of the preparing action, and then waits for replies. Participants reply with either *prepared* or *aborted* messages. If the coordinator receives a *prepared* message from each participant, it starts the committing phase below. If an *aborted* message is received, then the action must be aborted, and the coordinator informs the other participants via *abort* messages. The coordinator may also abort unilaterally if it does not receive responses from some participants, after suitable attempts to retransmit the *prepare* messages.

2. **Coordinator's Committing phase.** If all participants respond *prepared*, the coordinator writes a *committing* record, containing the names of the participants, to stable storage. At this point the action is committed. The coordinator then sends *commit* messages to all the participants (including itself), and waits for *committed* messages in response. When all have responded, the coordinator writes a *done* record to stable storage, and the two-phase commit is complete.

3. **Participant's Prepare phase.** When a participant receives a *prepare* message from the coordinator, it responds as follows. If the action is unknown at the participant (due, for example, to a crash), then the participant replies *aborted* to the coordinator. Otherwise, the current versions of all stable objects modified by the action (all stable objects listed in the MOS) are written to stable storage, read locks for all objects read by the action but not modified are released, and a *prepared* record is written to stable storage. The participant then replies *prepared* to the coordinator and enters the completion phase.

4. **Participant's Completion phase.** Once a participant has written the *prepared* record, it must await a verdict from the coordinator. When the participant receives a *commit* message, it writes a *committed* record to stable storage, releases write locks, replaces base versions with current versions, and then replies *committed* to the coordinator. If the participant receives an *abort* message instead, it writes an *aborted* record to stable storage, releases write locks, and discards current versions. If a participant has not heard from its coordinator it can query the coordinator to find out the outcome of the action. (An action identifier contains enough information that each participant knows who its coordinator is [13].)

As mentioned above, actions may be nested. In particular, handler calls run as subactions of the calling action. Subactions require extensions to the locking and version management rules given above (see [12]); for example, there is a separate version for each subaction that modifies an object. But these extensions are not significant as far as recovery is concerned because two-phase commit is only carried out when top actions commit. (Top actions

are not subactions of any action.) When a top action commits, only two versions exist for each object it modified: the base version records the state of the object before the action ran, and the current version records all changes made to the object by the action and its descendants. It is this current version that is written to stable storage.

3. The Log

Two main methods have been used in systems to organize stable storage: *logging* and *shadowing*. Like an accounting journal that is a chronological record of accounting transactions, a *log* [3, 7, 10] is a kind of append-only file used to record the versions of objects changed by an action as well as the stages of the two-phase commit protocol. The protocol information is needed to recognize when all values of data objects modified by a committing action have been written to the log, and when participants and coordinators must be recreated. For example, if a crash occurs before all modifications for an action have been written, then on recovery all modifications for the action will be discarded (and the action aborts).

In shadowing [7], storage is organized as a *map*, which associates objects with their actual versions in stable storage. As an action gets ready to commit, the new object versions are written to stable storage without destroying existing versions. When an action actually commits, these new versions are installed by making a new map that contains the pointers to them, writing the map to stable storage, and then switching from the old map to the new map in one atomic step. The old versions are then discarded. When an action aborts, the new versions are discarded and the map is untouched.

In a distributed system, a map alone is not enough for shadowing to work properly; information about the status of actions (prepared, committed, or aborted) is needed. For prepared actions, *intentions lists* [9] are also required. An intentions list contains the new piece of the map for the prepared action. A log might be used to maintain this information.

A cursory comparison of logging and shadowing would lead one to the conclusion that logging is faster during normal execution, since there is no need to update a possibly large map, but is slower during recovery, since an ever expanding log must be scanned in its entirety. There are, however, innumerable tricks and variations for each scheme, and ultimate performance depends heavily on the precise characteristics of the stable storage devices used. We will return to this issue at the end of the paper.

We have chosen to use a log-based mechanism for Argus, which we describe in the remainder of this section.

3.1. Log Abstraction Interface to Stable Storage

To avoid considering the details of implementing stable storage on top of conventional storage devices, we will simply assume the existence of a stable storage system that provides an efficient implementation of *stable logs*. A stable log resembles an array indexed by abstract objects called *log_addresses*.

The (stable) log abstraction provides the following operations:

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

2. *write(log, entry)*. This operation writes an arbitrary length entry to the log, and returns its log_address. The actual writing of the data to the stable storage may not have happened when this operation returns.

3. *force_write(log, entry)*. This operation forces an entry to the log, and returns its log_address. The current entry and all older entries have been written to stable storage when the operation returns.

4. *get_last(log)*. This operation returns the log address of the last entry that was forced to the log.

5. *read(log, log_address)*. This operation reads the entry at the log address and returns it.

6. *destroy(log)*. This operation destroys a log.

Each guardian has its own log; the Argus system remembers (in stable storage) the association between a guardian and its log.

3.2. Structure of the Log

Entries in the log can be classified as either *data entries* or *outcome entries*. A data entry contains a copy of a version of an atomic object; outcome entries indicate the stages of an action, such as whether an action has prepared, committed, or aborted. Figure 1 shows the formats of these entries.

Each outcome entry contains a log pointer, linking the entry to the previous outcome entry in the log. This reverse chain will be used during recovery to reconstruct the stable state. The outcome entries come in two varieties, one set -- *prepared*, *committed*, *aborted*, *base_committed*, and *prepared_data* -- for participants and the other set -- *committing* and *done* -- for coordinators.

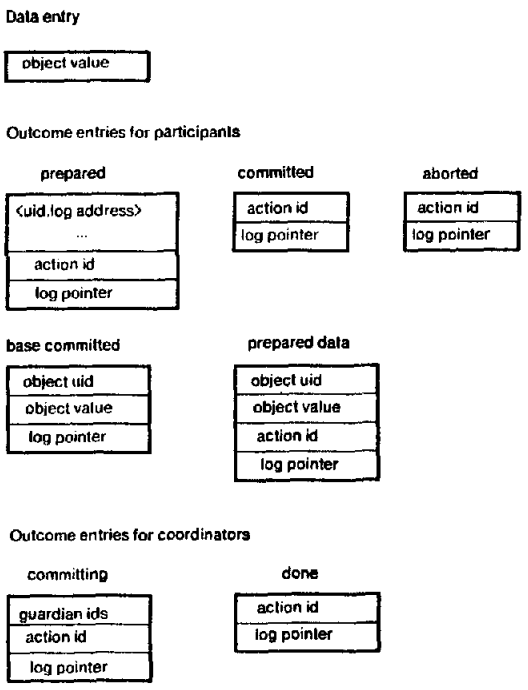


Figure 1: Format of log entries

A *prepared* outcome entry contains the action identifier of the preparing action, and a list of <uid, log address> pairs, where the uid is an identifier for an atomic object and the log address points to a data entry containing a copy of the object's version. This list is like the intentions list or partial map in a shadowing scheme; there is a pair for each atomic object that was both modified by the action and accessible from the stable variables.

A *committed* outcome entry is written to a participant's log by the recovery system when an action has committed, and an *aborted* outcome entry is written when an action has aborted. These entries contain the action identifier of the completing action.

Two special participant outcome entries, *base_committed* and *prepared_data*, handle certain cases arising from the dynamic nature of the guardian's stable state. The use of these entries, which are combined data and outcome entries, will be explained in the next section.

A *committing* outcome entry is written to a coordinator's log by the recovery system when all participants have prepared; it includes a list of all participants. A *done* outcome entry is written when two-phase commit is complete. Both entries contain the action identifier of the completing action.

4. Writing Objects to the Log

Our recovery system is similar to others that use a log. Whenever a participant receives a message from a coordinator, it carries out the requested action. If the message is a prepare, it writes out data entries containing the current versions of all objects modified by the action, followed by the *prepared* outcome entry. (Some of these data entries may have been written already because of early prepare as discussed in section 4.5 below.) Later, when it receives a commit message it writes out a *committed* outcome entry; when it receives an abort message, it writes out an *aborted* outcome entry. The coordinator writes out a *committing* outcome entry when all participants are prepared, and a *done* outcome entry when all participants acknowledge receipt of the commit message.

The result of this approach is that more recent information occurs later in the log than older information. During recovery, then, the recovery system will process the log backwards. For each data entry, it considers the status of the action on whose behalf that entry was written; information about the action status will be encountered *before* the entry since it was written to the log *after* the entry. If the action has aborted, or has no status, the recovery system ignores the data entry. (An action may have no status because, for example, a crash occurred while the action was preparing.) If the action has prepared, that is, a *prepared* outcome entry has been processed but not a *committed* or *aborted* outcome entry, then the recovery system uses the data entry to restore the object's current version. It also grants a write lock on the object to the prepared action; granting this lock allows us to resume execution of the guardian before the prepared action terminates. If the action has committed, the recovery system uses the entry to restore the object's base version, provided the base version has not yet been restored; if the base version is already restored, the entry contains an older and, hence, obsolete version of the object, so it is ignored.

This section discribes how we write information to the log. First, we discuss how atomic objects are copied to stable storage, and how we maintain the sharing relationship among objects when they are stored in the log. Next, we discuss how the system determines which modified objects should be written to stable storage, and how we deal with atomic objects that dynamically enter and leave a guardian's stable state. Then we describe our general writing method, and how to do early prepare.

4.1. Copying Data

The method we use for copying atomic objects to stable storage works in an *incremental* fashion: each atomic object is copied to stable storage in a separate, atomic step. To make incremental copying possible, each atomic object contains, in addition to lock information and versions, a unique object identifier called a *uid*. These uids find their use in the log. They are stored in the intentions list in the *prepared* outcome entry (see Figure 1), and also support inter-object references: a version in the log can refer to an atomic object by using that object's uid.

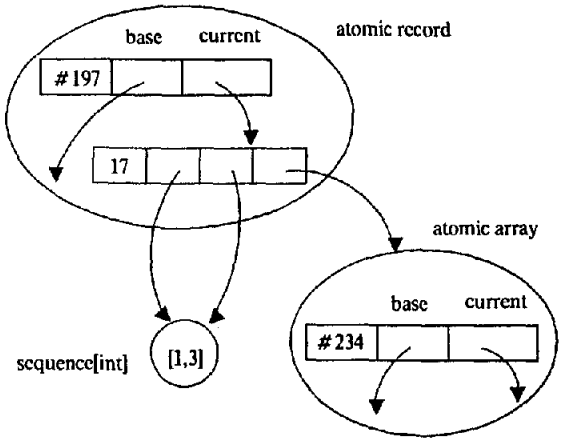
In volatile memory, objects are stored in a *heap*, and garbage collection is used to reclaim storage. Suppose object A contains object B as a component (for example, B is an element of an array A). The data storage for B is not physically contained within the storage for A; rather, B is an independent object and A contains the volatile memory address of B. (In actual practice, there are a few exceptions that arise out of performance considerations. Integers, in this example, would be contained in the storage for A, rather than existing as an independent object B with a pointer to it from A.)

Thus, a version of an atomic object may contain references to other objects, both atomic and non-atomic, which may in turn refer to other objects. An example of such an object is given in Figure 2a. The figure shows an atomic_record with four components; the first component is an integer, the second is a sequence of integers as is the third, and the fourth is an atomic array. The figure shows the current version of the object; notice that the second and third components refer to the same sequence.

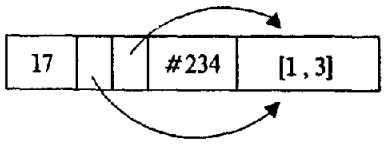
When a version is copied to the log, all non-atomic objects accessible from the version are also copied, but contained atomic objects are not. These contained atomic objects will be copied in a separate atomic step if they were modified by the preparing action.

In the copy on the log, contained non-atomic objects are contiguous, the intra-object references to non-atomic objects are replaced by relative offsets within the copy, and the references to other atomic objects are replaced by the uids of those objects. Figure 2b shows the log representation for the current version of the object in Figure 2a. Notice that the reference to the contained atomic object has been replaced by that object's uid.

Sharing relationships among the copied non-atomic objects are preserved. For example, the sharing of the sequence of integers is preserved in Figure 2b, and the use of uids for the atomic objects allows the sharing relationships among atomic objects to be preserved.



a. An atomic record in volatile memory



b. A copy of the current version of the record on the log

Figure 2: Object Formats

Because of the incremental nature of the copying, however, the sharing of a non-atomic object between several atomic objects is not preserved.

4.2. The Accessibility Set

As mentioned earlier, for each action, the Argus system keeps track of modified objects in a Modified Objects Set (MOS) for the action. The recovery system separates the objects in the MOS that are accessible from the guardian's stable variables from those that are inaccessible and only writes the accessible objects to the log.

Determining the accessibility of an atomic object could be accomplished by walking over the trees of objects accessible from the guardian's stable variables. To avoid race conditions, such a traversal must effectively block all user processes from running. This kind of approach would be extremely inefficient because there are likely to be many accessible objects. Instead, the recovery system maintains, for each guardian, an *accessibility set* (AS) of atomic objects accessible from the guardian's stable variables. The AS is implemented with a single bit in each atomic object; to determine whether an atomic object in the MOS must be written to the log, the recovery system merely checks this bit. The AS is initialized when the guardian is created by

performing an initial walk over the tree of objects accessible from the stable variables, setting the bit in each atomic object encountered. To maintain the AS, two problems must be solved: discovering when an object is *newly accessible*, and must be added to the AS, and discovering when an object is no longer accessible, and can be deleted from the AS.

Determining when an object is no longer accessible can only be accomplished by a full tree traversal from the stable variables. The current Argus implementation uses a stop-the-world, mark-sweep garbage collector to reclaim storage; recomputing the AS is done as part of the mark phase and requires essentially no additional overhead. Notice that this means the AS is really a superset of the atomic objects actually accessible. For each inaccessible object that remains in the AS, modifications to that object will result in unnecessary writes to stable storage. Usually, however, when an object is removed from the stable state it becomes completely inaccessible, and subject to reclamation at the next garbage collection. Such an object will not be modified because no program can access it. Therefore, waiting to recompute the AS until a garbage collection involves little unnecessary writing of inaccessible objects to stable storage.

4.3. Newly Accessible Objects

An atomic object is *newly accessible* if it is reachable from the stable variables but is not in the AS. The object may be newly created, or it may have existed for some time but was reachable only from volatile variables. Since the only way to make an object newly accessible is to modify some object that is already accessible, the recovery system can discover newly accessible objects simply by examining the other atomic objects encountered while copying accessible objects. In the process of copying an object, whenever a reference to an atomic object is replaced with the object's uid, the recovery system checks the referenced atomic object to see if it is in the AS. If not, the object is newly accessible and the recovery system must also copy it to the log, as well as add it to the AS. Copying the newly accessible object may, of course, result in finding more newly accessible objects.

Newly accessible objects introduce some new problems because the action that makes an object newly accessible may not have modified that object. One problem is that we may need to write the base version of the newly accessible object, instead of writing the current version of the object, which is what we always write for accessible objects. But a more significant problem is that if the preparing action did not modify the newly accessible object, then some other active action might be using that object concurrently. (Such concurrent use was not possible for accessible objects since the preparing action held a write lock on the object.) There are two situations to consider, each involving two or more concurrent actions. In the first situation, several actions make the object accessible; in the second, one action makes the object accessible, while another action modifies the object. The situations are illustrated with scenarios.

> **Scenario 1.** Suppose two actions, T_1 and T_2, both make some object, O, newly accessible through different paths, but neither modifies O itself. Suppose further that T_1 prepares before T_2. When T_1 prepares, O is found to be newly accessible, its base version is written to the log, and then O is added to the AS. When T_2 prepares, no version is written because O is already in the AS. Now let us consider what happens if T_1 aborts, T_2 commits, and the node crashes. On recovery, the version of O is ignored because it was written on behalf of an aborted action. O is still accessible via the commit of T_2, but there is no version of O for the recovery system to use.

To ensure that newly accessible atomic objects always have a base version at recovery, a special outcome entry, *base_committed*, is used. Whenever a newly accessible atomic object is encountered, its base version is copied to the log as part of a *base_committed* outcome entry. Such an outcome entry is not tied to a particular action (note that the entry does not contain an action identifier), and will be processed at recovery even if the action that caused it to be written ultimately aborts.

If a newly accessible object is in the preparing action's MOS, the object's current version is also written to the log as an ordinary data entry, to be included in the action's *prepared* outcome entry. The current version, however, must be copied in another case, as the following scenario illustrates.

> **Scenario 2.** Suppose action T_1 makes object O newly accessible, and that O has been modified (but not made accessible) by some other action, T_2. Suppose further that T_2 prepares before T_1. When T_2 prepares, no version of O is copied, since O is not accessible. When T_1 prepares, O's base version is copied but not its current version, since T_1 has not modified O. Now let us consider what happens if both T_1 and T_2 commit and the node crashes. On recovery, only the base version of O is restored, even though T_2 has committed. The modifications made by T_2 have been lost.

To avoid losing modifications, we use another special outcome entry. When a newly accessible object is write-locked by an action that has already prepared, the current version of the object is copied to the log as part of a *prepared_data* outcome entry. This entry contains the action identifier of the prepared action. At recovery, it will be treated as a (belated) addition to the *prepared* outcome entry of the specified action.

4.4. The General Writing Algorithm

We now present the full algorithm for preparing an action at a participant guardian. This algorithm runs when a prepare message is received at the guardian. The algorithm uses two new sets, maintained separately for each preparing action. An Intentions List (IL) is used to accumulate the <uid, log address> pairs for eventual inclusion in the *prepared* outcome entry. A Newly Accessible Objects Set (NAOS) is used to keep track of newly accessible objects.

The algorithm can be executed concurrently for multiple actions at the same guardian; in steps 2 and 3 below, the processing for an individual object must occur indivisibly with respect to concurrent prepares.

1. Create an empty NAOS and an empty IL.
2. ∀ object ∈ MOS do
   a. If object ∈ AS, write a data entry containing a copy of the object's current version, and add the <object uid, data entry log address> pair to the IL. For each atomic object encountered while copying the version, check the AS. If the object ∉ AS then the object is newly accessible, so add it to the NAOS for eventual processing.
   b. Delete the object from the MOS.
3. ∀ object ∈ NAOS do
   a. If the object is still ∉ AS (it may have been added by a concurrent prepare), write a *base_committed* entry containing a copy of the object's base version. For each atomic object encountered while copying the version, if the object ∉ AS then add it to the NAOS.
   b. Whether the object is still in the AS or not, if the preparing action has a write lock on the object, then write a data entry containing a copy of the object's current version, and add the <object uid, data entry log address> pair to the IL. For each atomic object encountered while copying the version, if the object ∉ AS then add it to the NAOS.
   c. If the object ∉ AS and some other action holds a write lock on the object, check the status of that action. If the locking action has already prepared, write a *prepared_data* entry containing a copy of the object's current version and the action identifier of the locking action. For each atomic object encountered while copying the version, if the object ∉ AS then add it to the NAOS. If the locking action is preparing concurrently, simply add the object to the NAOS of the locking action if that action's IL does not yet contain an entry for the object.
   d. Add the object to the AS and delete it from the NAOS.
4. Write a *prepared* outcome entry containing the IL, forcing it and all previous entries to the log with the *force_write* operation. The participant can then respond with a *prepared* message to the coordinator.

As mentioned earlier, the AS is initialized as part of guardian creation. Once the stable variables are initialized, they become read-only. At this point, the uids of the atomic objects that have been assigned to them are written to the log in a special record. These objects are then used as an initial NAOS, and a special version of step 3 of the algorithm above is run to initialize the AS and to write base versions of the stable objects to the log.

4.5. Early Prepare

The algorithm in the previous section delays all prepare activity until the *prepare* message is received from the coordinator. Once a handler call commits at a guardian, however, local processing on its behalf is over, so there is no reason why copying new versions of objects to the log cannot take place before the *prepare* message arrives. Early copying of versions to stable storage is called *early prepare*. Early prepare is done in background mode when the guardian is not busy. Ideally, by the time the *prepare* message arrives at the guardian, all modified and newly accessible objects have already been copied to the log. All that remains to be done is to write the *prepared* outcome entry itself.

Very few changes to the algorithm are needed to support early prepare. Rather than running the algorithm for each action, we can run it for each handler call made by the action. That is, we keep a separate MOS, IL, and NAOS for each handler call. The recovery system can start

running the algorithm (with the exception of the last step) as soon as a handler call completes. When the *prepare* message for the action is finally received, processing for each handler call that was part of the preparing action is run to completion, and then a merge of the resulting ILs is performed. Several handler ILs may contain entries for the same object, but with different object versions. The version used in the merged IL depends on which handler calls were retained by the action (an action may abort the effects of individual handler calls), and the subaction ordering of those committed handlers. Typically, the version for the last handler call will be used. Once the ILs have been merged, a *prepared* outcome entry is written to the log.

Early prepare may result in unnecessary information being copied to stable storage. For example, the action may ultimately abort, in which case all of the effort is wasted. If the action visits a guardian multiple times, and modifies the same object each time, multiple versions of the object may be written to the log. If the guardian has sufficient idle time, however, early prepare is almost always worth the effort, since it reduces the real-time delay imposed by two-phase commit.

5. Recovering Objects from the Log

After a crash, the recovery system reads the log backwards starting with the last outcome entry, reconstructing the stable state of the guardian as well as the action state. The recovery system restores the objects to the same state they were in before the crash. Three tables are used during recovery:

1. A participant action table (PT) maps action identifiers to participant action states:

   PT: action id → participant action state

   where participant action state is either *prepared*, *committed*, or *aborted*.

2. An object table (OT) maps object uids to object states:

   OT: object uid → object state

   The object state is either *partial* or *restored*, and contains the address of the object in volatile memory.

3. A coordinator action table (CT) maps action identifiers to coordinator action states:

   CT: action id → coordinator action state

   where coordinator action state is either *committing* or *done*. The committing state contains a list of the guardian identifiers of the participants.

To restore a logged version to volatile memory, the recovery system copies the version into volatile memory, adds a base address to each contained relative offset to obtain the new memory reference for a contained non-atomic object, and replaces each contained uid of an atomic object with a volatile memory reference to that object. Each contained uid is looked up in the OT. If it is present in the OT, then its volatile address is known, and its uid can be replaced with that address. Otherwise, a "null" object is created for it, and it is entered in the OT in the partial state. An object remains in the partial state until its base version is restored; the version (or versions) will be filled in as they are encountered later during recovery.

In the recovery algorithm described below, two kinds of object version restoration are used: *base* restoration and *prepared* restoration:

1. *base* restore. This is used to restore the base version of the object. Given an ⟨object uid, version log address⟩ pair, look up the uid in OT. If uid $\notin$ OT, the version is restored from the log and installed as the base version of the object, and the entry ⟨*uid*, *restored*, *vm address*⟩ is added to OT. If ⟨*uid*, *partial*⟩ $\in$ OT, the log version is restored as the base version, and the OT entry is changed to the *restored* state. If ⟨*uid*, *restored*⟩ $\in$ OT, the log version is ignored.

2. *prepared* restore. This is used to restore the current version of an object, and is therefore run on behalf of an action that has prepared but not yet committed. Given an ⟨object uid, version log address⟩ pair and an action identifier, look up the uid in OT. If uid $\notin$ OT, the version is restored from the log and installed as the current version of the object, the action is granted a write lock, and the entry ⟨*uid*, *partial*, *vm address*⟩ is added to OT. If ⟨*uid*, *partial*⟩ $\in$ OT, the log version is restored as the current version, and the action is granted a write lock. (⟨*uid*, *restored*⟩ $\in$ OT is not possible since we process the log backwards, and no action whose outcome entry appears in the log after the prepare record of this action can have used the object, since this action held an exclusive lock on it at the time of the crash.)

The general recovery algorithm can now be described.

1. Create an empty PT, OT, and CT. Volatile memory is empty.

2. Read the log backwards, starting with the last outcome entry in the log. Process each entry as follows.

   a. *done* outcome entry. Insert ⟨*aid*, *done*⟩ in the CT.

b. *committing* outcome entry. If ⟨*aid*, *done*⟩ ∈ CT then ignore the entry. Otherwise, insert ⟨*aid*, *committing*(*gids*)⟩ into the CT, where *gids* are the guardian identifiers of the participants as given in the outcome entry.

c. *committed* outcome entry. Insert ⟨*aid*, *committed*⟩ in the PT.

d. *aborted* outcome entry. Insert ⟨*aid*, *aborted*⟩ in the PT.

e. *prepared* outcome entry. Look up the *aid* in the PT.

   i. ⟨*aid*, *committed*⟩ ∈ PT. ∀ ⟨uid, log address⟩ pairs, do a *base* restore.

   ii. ⟨*aid*, *aborted*⟩ ∈ PT. Do nothing.

   iii. *aid* ∉ PT. Insert ⟨*aid*, *prepared*⟩ in the PT, and ∀ ⟨*uid*, *log address*⟩ pairs, do a *prepared* restore.

f. *base_committed* outcome entry. Do a *base* restore with the ⟨uid, log address⟩ pair.

g. *prepared_data* outcome entry. Look up *aid* in the PT.

   i. ⟨*aid*, *committed*⟩ ∈ PT. Do a *base* restore with the ⟨uid, log address⟩ pair.

   ii. ⟨*aid*, *aborted*⟩ ∈ PT. Do nothing.

   iii. *aid* ∉ PT. Do a *prepared* restore with the ⟨uid, log address⟩ pair.

3. The special log record identifying which objects are assigned to the stable variables is read, and the stable variables are initialized. The recovery system traverses the objects accessible from the stable variables, recreating the AS.

4. The PT and CT are returned to the Argus system, to recreate two-phase commit activities.

We now present an example of recovery. Suppose we have the log depicted in Figure 3. In this figure the log grows to the right. The symbols in the log have the following meaning. T_1 and T_2 are action identifiers. 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. L_1 and L_2 are log addresses. The arrows are back pointers either to data entries or to other outcome entries.

The following is a step-by-step explanation of how the objects in the log depicted in Figure 3 are recovered.

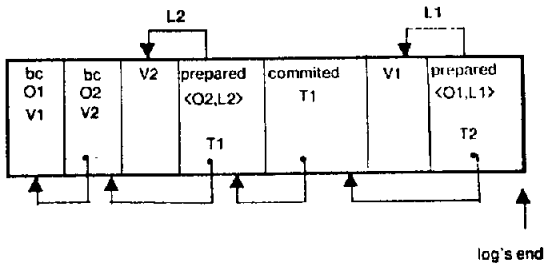


Figure 3: Log after the prepare phase

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

1. For outcome entry ⟨prepared, ⟨O_1, L_1⟩, T_2⟩ the recovery system checks the PT for T_2 and enters ⟨T_2, *prepared*⟩ into the PT.

   For the ⟨O_1, L_1⟩ pair the recovery system checks the OT for the uid, finds it is not there, follows the L_1 pointer to the data entry, copies the object version V_1 to volatile memory as the current version for an atomic object, and grants T_2 a write lock. The recovery system enters ⟨O_1, *partial*, vm address⟩ into the the OT and follows the log pointer to the previous outcome entry.

2. For outcome entry ⟨committed, T_1⟩ the recovery system enters ⟨T_1, *committed*⟩ into the PT, and follows the log pointer to the previous outcome entry.

3. For outcome entry ⟨prepared, ⟨O_2, L_2⟩, T_1⟩ the recovery system looks up T_1 in the PT and finds the ⟨T_1, *committed*⟩ entry.

   For the ⟨O_2, L_2⟩ pair the recovery system looks up O_2 in the OT, finds it is not there, follows the L_2 pointer to the data entry, and copies the object version V_2 to volatile memory as the base version of the object. The recovery system enters ⟨O_2, *restored*, vm address⟩ into the OT, and follows the log pointer to the previous outcome entry.

4. For outcome entry ⟨base_committed, O_2, V_2⟩, the recovery system looks up O_2 in the OT. Since the object already appears in the OT with state *restored*, the most current version of the object has already been copied, so the recovery system ignores the entry. The recovery system follows the log pointer to the previous outcome entry.

5. For outcome entry ⟨base_committed, O_1, V_1⟩, the recovery system looks up up O_1 in the OT. Since the state of the object is *partial*, the recovery system copies V_1 into volatile memory

as the base version of the object and changes the object state to *restored*. The log pointer is null, so there are no more outcome entries to consider.

6. Housekeeping the Log

We have seen that the log is a repository of all the atomic objects that were ever part of the guardian's stable state during the guardian's lifetime. As time goes on, the log will eventually become quite large. Although the recovery system need not read all data entries, it must read *every* outcome entry to reconstruct the stable state. If the log is too large, recovery will be unacceptably slow. Furthermore, the recovery system cannot determine the stable state until all objects have been recovered from the log. As new objects replace older objects in the stable state, the total number of objects restored from the log may be much larger than the number of objects actually contained in the final stable state, perhaps even more objects than will fit in volatile memory.

One way to reduce the size of the log is to produce periodically a *snapshot* representing the stable state of a guardian and to write that snapshot into a new log. This new log replaces the old log when the snapshot is complete. For example, a snapshot could be taken immediately after recovering from a crash if the log is large compared to the final size of the stable state. For very active guardians running on reliable nodes, however, waiting for a crash to take a snapshot is inadequate, and it is undesirable to force the guardian to crash just to clean up the log. Stopping the guardian to take a snapshot is just as bad as crashing it, because activity may be suspended for an undesirable length of time. Therefore, we describe a dynamic snapshot scheme below. This scheme does not affect the recovery algorithm at all, and requires only minor changes to the writing algorithm.

A snapshot takes place in two stages. First, the recovery system copies the accessible objects into a new log. Second, it copies to the new log all outcome entries (together with their associated data entries) written to the old log since the snapshot began. The snapshot is performed by a process running in parallel with other system and recovery operations, synchronizing with them as needed. The snapshot process generally runs only when the guardian is otherwise idle, although if little progress is being made on the snapshot, it may be necessary to give the process more time.

When the snapshot process first starts up, it determines the earliest log address recorded in the ILs of actions being prepared (or early prepared) by the recovery system, and remembers that address. The snapshot process also creates a new, empty accessibility set (the snapshot AS), and a new, empty log (the snapshot log). It then traverses the graph of atomic objects accessible from each of the guardian's stable variables. For each atomic object it encounters, it checks whether the object is already in the snapshot AS. If not, it adds the object to the snapshot AS, and writes a *base_committed* entry containing the base version of the object to the snapshot log.

When the snapshot process has completed its traversal of the stable state, it enters the second stage of the snapshot, to deal with entries written to the normal log while the snapshot was being taken. These entries must be copied over to the snapshot log. Starting with the entry at the log address recorded when the snapshot process first started, the snapshot process reads each entry (in forward order) from the normal log and writes a new copy of the entry to the snapshot log. For each data entry copied over to the snapshot log, the snapshot process maintains a mapping from log address in the normal log to corresponding log address in the snapshot log. When a *prepared* outcome entry is read from the normal log, the snapshot process must update each contained <uid, log address> pair with the new snapshot log address before writing the entry to the snapshot log.

When all entries in the normal log have been copied, the normal recovery system is suspended while the normal log is replaced with the snapshot log, and all log addresses in all ILs of preparing actions are mapped to their new addresses. The recovery system then resumes writing log entries to the new log.

This snapshot algorithm can result in unnecessary copying of data. For example, the first stage may write a base version, only to have that version supplanted by a version from an action that commits while the snapshot is being taken. The amount of extra copying is proportional to the amount of processing that occurs against the old log while the snapshot takes place. We assume that the extra copying is not significant compared with the total size of the stable state.

7. Discussion

We have described a log-based organization of stable storage, and presented algorithms for writing objects to the log and recovering objects from the log after a crash. A significant feature of these algorithms is the handling of a dynamic, implicitly changing stable state via a guardian's Accessibility Set. We introduced an early prepare

mechanism to speed up two-phase commit, and a snapshot mechanism to speed up crash recovery.

We believe this is a reasonably efficient method for organizing stable storage. In the expected normal case, assuming enough idle time for early prepare to finish, the total delay a user process incurs for two-phase commit is approximately one message round trip time (prepare and prepared messages) plus the time for two stable storage writes (prepared and committing entries); no delay accrues from phase two of the protocol, as it can be carried out in background. This is about the best one can hope for with any method. Although the log is biased somewhat in favor of normal processing at the expense of recovery speed, the snapshot mechanism should provide recovery times reasonably close to those possible with a shadowing scheme, again assuming guardians have sufficient idle time.

Argus is running on VAX-11/750 processors under Berkeley Unix 4.2. The basic writing and recovery algorithms have been implemented and are in use in the Argus system, although we are currently using conventional disks instead of true stable storage. Early prepare and snapshots are not yet implemented.

	Current time conventional disks	Estimated time stable storage
message round trip	35	15
one block disk write	60	20
total user delay	265	95

Figure 4: Current and Estimated Performance Figures

Figure 4 presents our current and estimated performance figures. A message round trip time is approximately 35 milliseconds; measurements show that more than half of this time can be attributed to unnecessary kernel overhead. Our measurements show that a typical one block raw disk write takes about 20 milliseconds; however, we are using a network-based disk server, with a write time of about 60 milliseconds. With this configuration, two-phase commit delays a user process a minimum of 155 milliseconds, but because Unix lacks non-blocking raw disk writes, the writes done in "background" in phase two of the protocol block the entire guardian, so the delay with sustained activity is closer to 265 milliseconds. The estimated delay of 95 milliseconds assumes one round trip message and four disk writes, two each for writing the prepare and committing records. If we are willing never to overwrite disk pages as is done, for example, in Swallow [16, 17], then the delay can be reduced to 55 milliseconds.

During recovery the system must read every outcome entry, but may only look at a fraction of the data entries in the log. To reduce the number of unnecessary reads during recovery, it might be better to keep two separate logs, one containing information about actions (outcome entries), and the other containing information about data (data entries). We have merged the outcome and data entries into one log to minimize the number of sequential stable storage writes, under the assumption that writes are expensive. Suppose, however, one built a large *stable buffer* out of highly reliable direct access memory, so that disk operations could generally take place in background. Write times are then almost negligible, perhaps making other stable storage organizations more attractive.

We have organized stable storage under the assumption that volatile memory has been discarded, or at least cannot be trusted, after a crash, which is certainly true of most operating systems. If, however, we were to design an operating system kernel specifically for Argus, we might use the virtual memory system as part of the stable storage system, in the following way. Notice that the version management used for atomic objects in volatile memory is essentially a shadowing scheme: the base version is the shadow and committing an action installs a new shadow. If we assume that after a crash the backing store portion of virtual memory on disk is intact, then recovery can be done starting with the object versions in virtual memory and consulting the log to bring the objects up to date. Changes of uncommitted actions can be undone simply by discarding the current versions of objects; for the prepared actions we must retain the current versions. Since some modified pages in volatile memory may not have been paged out to backing store before the crash, we cannot be sure whether needed base and current versions were written to disk. To keep backing store approximately up-to-date, we can create checkpoints, somewhat like those in System R [7], by periodically suspending activity long enough to flush all modified pages containing stable objects and their base versions to backing store. After a crash only the part of the log after the checkpoint, plus data entries before the checkpoint for actions whose prepare or commit record appears after the checkpoint, will have to be processed.

Such a method is likely to be faster for recovery than our current scheme. One problem arises, however: what happens if a crash occurs during garbage collection? Such a crash is not a problem with our current scheme because virtual memory is discarded after the crash. However, if virtual memory is retained, it must be possible to restore objects from it even if it was being reorganized by garbage collection when the crash occurred.

Many garbage collection schemes make virtual memory unreadable while they are running. Our own scheme is one of these; we use a mark-sweep collector that stores information in the objects themselves. Other schemes, such as a copying garbage collector, do not destroy virtual memory, but these appear to be the less efficient than our scheme given our current hardware constraints. Such schemes are safe because they consist of a sequence of atomic steps. We intend to explore alternative garbage collection methods that are both atomic and efficient as part of our future research.

References

1. Arens, Gail C. "Recovery of the Swallow Repository". Tech. Rep. MIT/LCS/TR-252, M.I.T. Laboratory for Computer Science, January, 1981. Master's thesis.

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

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

4. Davies, Charles T. "Recovery Semantics for a DB/DC System". *Proceedings of the 1973 ACM National Conference*, 1973, pp. 136-141.

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

6. 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 (November 1976), 624-633.

7. Gray, James N. "Notes on Database Operating Systems". In *Lecture Notes in Computer Science 60*, Goos and Hartmanis, Eds., Springer-Verlag, Berlin, 1978, pp. 393-481.

8. Israel, Jay E., Mitchell, James G., and Sturgis, Howard. "Separating Data from Function in a Distributed File System". Tech. Rep. CSL-78-5, Xerox Palo Alto Research Center, September, 1978.

9. Lampson, Butler W. and Sturgis, Howard E. "Crash Recovery in a Distributed Data Storage System". Xerox Palo Alto Research Center,Palo Alto, California (1976). Unpublished paper. This 1976 paper is the first to mention the notion of intentions lists.

10. Lampson, Butler W. and Sturgis, Howard E. "Crash Recovery in a Distributed Data Storage System". Xerox Palo Alto Research Center,Palo Alto, California (April, 1979). Unpublished paper. This later paper mentions nothing about intentions lists.

11. Lampson, Butler W. Atomic Transactions. In *Lecture Notes in Computer Science, Volume 105: Distributed Systems: Architecture and Implementation*, Goos and Hartmanis, Eds., Springer-Verlag, Berlin, 1981, ch. 11, pp. 246-265.

12. Liskov, Barbara H. and Scheifler, Robert W. "Guardians and Actions: Linguistic Support for Robust Distributed Programs". *ACM Transactions on Programming Languages and Systems* 5, 3 (July 1983), 381-404.

13. Liskov, Barbara H. "Overview of the Argus Language and System". Available as Programming Methodology Group Memo 40, M.I.T. Laboratory for Computer Science, February, 1984.

14. Moss, J. Eliot B. "Nested Transactions: An Approach to Reliable Distributed Computing". Tech. Rep. MIT/LCS/TR-260, M.I.T. Laboratory for Computer Science, June, 1981. Ph.D. thesis.

15. Oki, Brian M. "Reliable Object Storage to Support Atomic Actions". Tech. Rep. MIT/LCS/TR-308, M.I.T. Laboratory for Computer Science, May, 1983. Master's thesis.

16. Reed, David P. and Svobodova, Liba. "Swallow: A Distributed Data Storage System for a Local Network". In *Local Networks for Computer Communication*, West, A. and Janson, P., Eds., North Holland Publishing Company, 1981, pp. 355-373.

17. Svobodova, Liba. Management of Object Histories in the Swallow Repository. Tech. Rep. MIT/LCS/TR-243, M.I.T. Laboratory for Computer Science, July, 1980.

18. Weihl, William E. and Liskov, Barbara H. "Specification and Implementation of Resilient Atomic Data Types". Available as Computation Structures Group Memo 223, M.I.T. Laboratory for Computer Science, December, 1982. This memo contains a detailed discussion of mutex.