Disconnected Actions: An Asynchronous Extension to a Nested Atomic Action System

by

Boaz Ben-Zvi

January 1990

This research was supported in part by the Advanced Research Projects Agency of the Department of Defence, monitored by the Office of Naval Research under contract N00014-83-K-0125, and in part by the National Science Foundation under grant DCR-8503662.

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

Disconnected Actions: An Asynchronous Extension to a Nested Atomic Action System

by

Boaz Ben-Zvi

Submitted to the Department of Electrical Engineering and Computer Science
on January 19, 90, in partial fulfillment of the
requirements for the degree of

Master of Science

Abstract

Nested transactions, a generalization of atomic transactions, provide a uniform mechanism for coping with failures and obtaining concurrency within an action. Execution of a nested action is synchronized with its creating action by halting the creator until the execution of its nested action terminates.

This thesis proposes a new kind of a nested action called a *Disconnected Action* (DA, for short) that runs asynchronously with its creating action. Disconnected actions allow additional work to be done in parallel with the rest of the work in a nested atomic action system. Work done by a DA improves the performance of its creating action, but is not needed for the correctness of this action.

This thesis describes how DAs semantics are achieved; existing mechanisms, such as the two-phase commit, are modified, and serial behavior of DAs is ensured by a mechanism that uses timestamps to enforce static creation order, and tables to monitor and control dynamic serialization order of concurrent DAs. The thesis also proposes a technique that allows an action using DAs to commit in spite of some failures.

Thesis Supervisor: Barbara H. Liskov
Title: N.E.C. Professor of Software Science and Engineering

Keywords: Distributed systems, Transactions, Nested transactions, Disconnected nested transactions, Concurrency control.

Acknowledgments

Many people have helped me in the long process of converting an initial idea to a detailed thesis. I am in debt to the following people:

Barbara Liskov, my thesis supervisor, who read countless drafts of the thesis tirelessly. Her observant eye caught every flaw, and her guidance helped me bring the work into a steady state. Barbara has also helped me convert my unique version of the English language to something the natives understand.

My office mates during the long way: Mark Day, the Argus hacker; Rivka Ladin, the non-hacker; Nobuyuki Saji, who showed me the guts of Emacs; and the newcomers, Qin Huang and Barbara Gates, who helped create an atmosphere of work in the office while I was finishing the thesis.

Members of the Programming Methodology Group, in past and present, who were always attentive to my questions: Elliot Kolodner, Bill Weihl, Sanjay Ghemawat, Sharon Perl, Carl Waldspurger, Paul Johnson, Dorothy Curtis, Earl Waldin, Brian Oki, Robert Gruber, Gary Leavens, Deborah Hwang, Liuba Shrira, Andrew Xu, Wilson Hsieh, Jeff Cohen, Anthony Joseph, and Maurice Herlihy.

Other members of the Lab for Computer Science in MIT, whose friendship helped me through: Dan Jackson, Ken Goldman, Hagit Attiya, Yishay Mansour, Arie Rudich, Tim Shepard, Baruch Awerbuch, Shlomo Kipnis and many others whose names slipped my memory in the haste of finishing the thesis.

People of the IBM Haifa Scientific Center, who helped me transform from an undergraduate student to a researcher. Special thanks to Uri Shani, Ron Pinter and Miki Rodeh.

All the members of CSL at the Xerox Palo Alto Research Center, with whom I spent a fruitful summer. In particular: Doug Terry, who helped me understand systems better; Ralph Merkle, who proved that people of theory have something to say about systems; Larry Masinter, who showed me that hacking is actually an art.

Finally, special thanks to my family. The consideration I received from Janna, my wife, had helped me devote my time and energy to my work. My son, Eran, deserves his share of gratitude for preventing me from waking up late each morning.

Contents

List of Figures

Chapter 1

Introduction

The main challenge facing the design of reliable distributed computing systems is maintaining data integrity in the presence of concurrency and failures. Atomic transactions, or actions, are a widely accepted solution to these two problems; they are serializable and recoverable, thus hiding concurrency and failures.

Nested transactions ([Reed 1978, Moss 1981, Liskov & Scheifler 1983]) are a generalization of the model of atomic transactions. Nested transactions, or subactions, provide a uniform mechanism for coping with failures and obtaining concurrency within an action. Execution of actions in a nested action model is synchronized in a manner analogous to making procedure calls in a programming language; an action that created a subaction (or a group of concurrent subactions) halts until the execution of its subaction terminated.

This thesis proposes a new kind of a nested action called a *Disconnected Action* (DA, for short) that, unlike a regular subaction, runs asynchronously with its creating action. Disconnected actions allow additional work to be done in parallel with the rest of the work in a nested atomic action system. Work done by a DA improves the performance of its parent action, but is not needed for the correctness of this action.

Disconnected actions can be used to perform benevolent side effects within the scope of their action. They can be used to update caches or improve representation of abstract data objects. For example, consider an object representing a rational number (a fraction), with a pair of numbers as its internal representation, for numerator and denominator. An action may create a subaction to update the object, followed by a DA to bring the fraction into canonical form.

The rest of this chapter is organized as follows: A motivating example for the use of disconnected actions, a replicated file, is presented in detail below. Section 1.2 overviews some basic work in the field of nested atomic actions. The chapter concludes with an overview of the rest of the thesis.

1.1 Replicated File System – an example

Assume a replicated file system that employs a simple voting scheme (like those in [Thomas 1979, Gifford 1979]). There are N sites, each containing a replica of the file and a version number. A file is updated by assembling a *write quorum* of W sites, and read by a *read quorum* of R sites. In order to ensure *one copy serializability*[1], the quorums should satisfy two constraints:

$$W + R > N \quad and \quad 2W > N$$

that is, the read quorum and write quorum should include at least one common site to ensure that a read operation will see effects of previous write operations, and having at least one common site for every two write quorums ensures serial order of write operations.

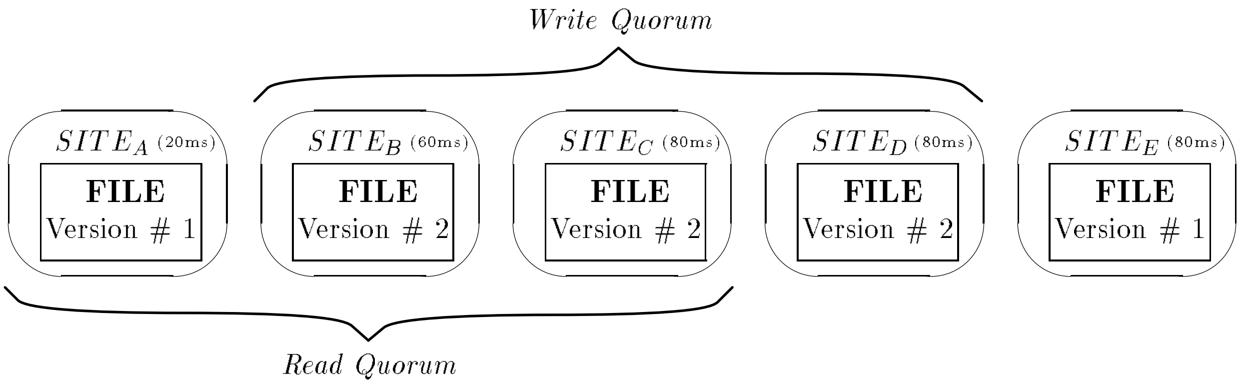


Figure 1-1: An example for a voting scheme.

Figure 1-1 shows an example for a replicated file system using the voting scheme described above, with five sites and read or write quorums of three sites (i.e., $N = 5, W = 3, R = 3$).

[1]Meaning: the user should not be able to observe the replication. The replicated file should behave like a single copy.

Each site has its copy of the file and a version number. The up-to-date copies have the highest version numbers. The figures in parenthesis indicate the latency from the reader's site.

Since files usually are large in size, a file is read in two phases. First, a read quorum is assembled, and the version numbers of the files in this quorum are read. The reader can tell, based on these numbers, which sites in its quorum have up-to-date replicas of the file. In the second phase, the file itself is read from **one** of these up-to-date sites, preferably from a site that has the lowest latency (the local copy, for example).

Though the reader is guaranteed to get the current version of the file, its choice in selecting a site to read from may be limited, and its preferred site may not have an up-to-date version. In the above example, assume that all the sites started with a consistent copy of the file with version number 1, and a write operation updated the files of a quorum of sites B, C and D, incrementing their version numbers to 2. A following read operation, using a quorum of sites A, B and C, finds the current version in sites B and C only. Site A, which has the lowest latency, can not be used since its version is not up-to-date.

We would like to have the writer write to all sites, but not be delayed longer than necessary. This can be accomplished by having the writer write to some write quorum, and then continue and have the rest of the replicas updated "in the background". The work done in the "background" should be consistent with the semantics of transactions. That is, the effects of the "background" updates should disappear if the updating action or one of its ancestors aborts, and become permanent when the topaction commits.

Disconnected actions can serve this purpose. A writer trying to update a replicated object can write to a write quorum using regular subactions, and after they commit create disconnected actions to write to the rest of the replicas (those not in the write quorum). The writer continues to run immediately after creating those DAs; their fate has no effect on the correctness of the update operation.

Better performance of the write operation is also expected in the scheme above, because replicas are more likely to be up-to-date in the scheme that uses DAs without additional time cost. In a scheme without DAs, which updates only a write quorum, some replicas may be far behind. With large files, when a write operation that modifies part of the file (e.g., an append operation) finds such an archaic replica in its quorum, it has to do more work: it has to read more records from an up-to-date replica and write those records to the archaic one.

1.2 Related work

The idea of nested atomic actions was initially proposed, as *spheres of control*, in [Davies 1973, Bjork 1973]. The first detailed design for a model that uses nested atomic actions was developed by Reed ([Reed 1978]). Reed proposed a multi-version, timestamp-based algorithm to ensure serialization of concurrent actions. A locking-based model of a nested atomic action system was developed by Moss ([Moss 1981]).

Several research projects have designed and implemented a nested atomic action system: Argus ([Liskov & Scheifler 1983]) uses Moss's model with small modifications (e.g., no distinction between lock-holding and lock-retaining). The Camelot project ([Spector, *et al.* 1987]) uses a model of nested actions similar to Argus', but with several differences (e.g., a remote calls does not require creating a new subaction). And the Clouds project ([Allchin & McKendry 1983]) also uses a similar model with some variations (e.g., using a top-level commit protocol to commit subactions at any level, hence subactions are not as "cheap" as in Argus). Clouds also uses different levels of object locking, and leaves the choice of level to the programmer.

Our work has some similarities to other works that enhanced the nested action model in some ways. In [Walker 1984], Walker proposed algorithms that piggybacked information on existing messages in order to detect orphaned computations. In [Perl 1987], Perl also uses similar techniques to reduce the need for lock query messages by propagating commit and abort information with existing messages (she calls it *eager diffusion*).

There are two basic techniques for serializing concurrent actions: timestamping ([Reed 1978, Aspnes, *et al.* 1988]) and two-phase locking ([Eswaran, *et al.* 1976]). Our technique of adding timestamps to the locking-based model can be considered as a hybrid serialization technique. Hybrid protocols are discussed in [Bernstein & Goodman 1981]; their protocols are general methods of synchronization that handle each of the read-write or write-write conflicts with separate two-phase locking or timestamp techniques. Another hybrid technique is discussed in [Weihl 1984]; it proposes the use of (static) creation-order timestamps for read-only activities and (dynamic) commit-order timestamps for update activities.

Our technique of explicit control of serialization (using constraint tables) relates to the idea of *conflict graphs*, also discussed in [Bernstein & Goodman 1981]. However, the conflict graphs there are used only as a helping tool for a scheduler to improve performance of a certain timestamp technique. Our notion of similarity between deadlock detection and serializability of concurrent actions has also been noted in [Zhao & Ramamritham 1985].

Much work has been done on the design of commit protocols for top-level actions. [Gray 1978] describes the classic two-phase commit protocol, and some variations like "nested two phase commit protocol", where the participants are ordered and every one communicates with the next one. Other variations include non-blocking protocols ([Skeen 1981]) and three-phase protocols (like two-phase, but with an initial phase that tells the participant to "prepare for prepare").

Our work proposes a model that tries to commit the transaction in spite of failures. In [Gifford & Donahue 1985], a different way is tried to avoid aborting a long-lived atomic transaction due to failures. They propose breaking the atomic transaction into independent atomic actions; thus a failure causes only a few of the actions to abort, and they could be retried.

1.3 Road map

The reminder of this thesis is organized as follows:

- Chapter 2: Describes the current model of computation and defines terminology and notation.
- Chapter 3: Describes how disconnected actions fit into the current model.
- Chapter 4: Describes how concurrency control is handled in a system using DAs. Two mechanisms are described, timestamping and conflict tables, that ensure serialization of DAs.
- Chapter 5: Describes how the two phase commit protocol should be modified to handle a system that uses DAs.
- Chapter 6: Proposes methods to enable transactions to commit in spite of failures by taking advantage of the semantics of DAs.
- Chapter 7: Summarizes the work and proposes future additions.

Chapter 2

The System Model and Definitions

In this chapter we describe our model of computation, which is basically the high-level model employed by the Argus programming language and system ([Liskov 1984, Liskov, *et al.* 1987b]).

2.1 The underlying system

We view the low-level system as a distributed collection of nodes (i.e. computers) connected by a communication network. The nodes communicate only by sending messages over the network. A node is typically a processor with some fast volatile memory memory and slower, non-volatile, stable[1] storage.

Any component of the system may fail, but the likelihood of multiple simultaneous failures is small. Messages sent over the network may be lost, corrupted or duplicated. We assume that an underlying Datagram protocol (e.g., [Postel 1980, Boggs, *et al.* 1979]) ensures delivery of uncorrupted messages. When that Datagram protocol fails (i.e., *network partition*), the high-level system has to be notified.

Nodes may fail, but we assume that they eventually recover. When a node fails (i.e., *crashes*), the contents of its fast volatile memory are lost, but its stable storage remains intact. We assume fail-stop processors ([Schlichting & Schneider 1983]) in the nodes; that is, a failed processor does not send random messages or write arbitrarily to its storage.

[1]Some researchers make a distinction between non-volatile storage (e.g., single local disk) and stable storage (e.g., double-disk scheme ([Lampson & Sturgis 1976]), but in this thesis we shall treat the two as one.

2.2 The current model

This section describes the model of computation that is used currently by the Argus system ([Liskov, *et al.* 1987b]), without any changes or optimizations (e.g., as suggested in [Perl 1987]). Some small modifications are introduced, however, both to the current implementation and the terminology to make it easier to extand the model to support our new algorithms and protocols. When the algorithms are not straightforward , we describe them using a Clu-like notation ([Liskov & Guttag 1986]).

2.2.1 Sites

A *site*, or *guardian* in Argus, is an abstraction for a node[2]. The high-level system is a collection of sites, each encapsulating several resources. The site keeps its state in internal data objects, not accessible from the outside; the only way to manipulate these objects from the outside (e.g., by other sites) is through *handler calls*, similar to the way operations manipulate the internal representation of an abstract data object.

Processes (*threads*) are created inside the site to carry out handler calls and background activity. Threads may share and manipulate the site's data objects directly. Each thread has its own execution stack, but compound data objects are allocated from a heap. All threads are lost when their site crashes.

A site has two kinds of objects: *stable* objects and *volatile* objects. Stable objects are written to stable storage after being modified by top-level actions (see below); therefore stable objects survive site crashes with high probability. Volatile objects are lost when their site crashes; they are used for recording redundant information (e.g., indices) or information that can be discarded after a site crash (e.g., internal state of a thread). Objects are handled internally at site; no reference to an object can be sent to another site (at the system level).

2.2.2 Actions

Atomic *actions* are a mechanism for maintaining data consistency in the presence of concurrency and failures. An atomic action transfers the state of the system from one consistent state (i.e.,

[2] Though several sites may exist at a single node.

a state that satisfies some invariants) to another consistent state. Atomic actions (actions, for short) were originally used in centralized systems, but are particularly important in distributed systems, where concurrency is real[3] and failures may be partial. Distributed programs may run simultaneously, each at several sites, share data objects and have to cope with failures such as site crash and network partition. Actions are the basic programming tool to build distributed programs that cope with concurrency and failures.

Actions are *serializable* and *recoverable* (*total*). Serializability means that when actions are executed concurrently, the effect is as if they were run sequentially in some order. This feature allows programmers using actions to ignore concurrency to a great extent[4]. Recoverability means that an action either completes successfully or is guaranteed to have no effect (i.e., "all or nothing" semantics). An actions that completes is said to *commit*; otherwise, the action *aborts*. (*Terminate* is a general term for either commit or abort.) The recoverability feature serves as an automatic checkpoint mechanism for programmers, saving the trouble of writing a "rollback" code to undo work of a program that can not complete due to failures.

Atomicity is implemented in our model with the use of *atomic objects*. This mechanism synchronizes accesses of actions to shared objects to provide serializability, and provides a way to recover the old state of any objects modified by an action that aborts.

2.2.3 Nested Actions

Our model generalizes atomic actions to be *nested*. A nested action is an action that is started inside another action, and may itself start more nested actions, forming an action tree of arbitrary levels. Nested actions provide action semantics within an action; they act as a checkpoint mechanism within an action, and also provide concurrency within an action.

We use the following terminology for actions in our model: a *subaction* is a nested action. A *topaction* is an action that may have subactions, but is not itself a subaction (i.e., the top of the action tree). An *action* is any single action (i.e., either subaction or topaction). The term *transaction*, commonly used in systems without nested actions instead of the term *action*, is used here to refer to the "whole action tree", instead of a specific action in the tree. For example,

[3] Unlike timesharing centralized systems that use global data for synchronization.
[4] Some concurrency related problems, such as deadlocks, may still need a programmer's consideration.

we may talk about an object modification done by the transaction, instead of specifying exactly which subaction (or topaction) did it.

Tree terminology is used to refer to relations between actions. An action A creating a nested action B is a *parent*, while B is a *child*. We also use the term *descendant*, meaning any of the action itself, its children, their children, and so on. For example, A, B and any child C of B are all descendants of A. Similarly the term *ancestor* is used in the other direction (e.g., A,B and C are all C's ancestors). The prefix *proper* is used to exclude the action itself. For example, only B and C are A's *proper descendants*, and only A and B are C's *proper ancestors*. The term *relative*[5] is used to relate one action to another; both must be from the same transaction. The term *least common ancestor* (or LCA) of two actions, A and B, refers to the lowest action in the tree that is an ancestor of both A and B.

An action may create one nested action at a time, or a set of subactions that run concurrently. In both cases the action is synchronized with its children by being blocked until the active children terminate. Children created by the same action are related to each other as *siblings*.

Subactions that belong to the **same** set of concurrent subactions are related to one another as *concurrent siblings*. Subactions from different sets (of the same parent) are not concurrent siblings. We use the term *relation* to refer to one of these two subaction creation orderings, serial or concurrent. Unlike some previous works, this thesis does not regard concurrency as an inherent feature of the action but as its (ordering) relation to another relative action[6]. Furthermore, we found it useful to think about (the implementation of) a set of concurrent actions as a single special subaction (*concurrent-set action*) that does no real work (like the call-action described below), but only creates a set of subactions that run concurrently and commits after all the concurrent subactions in its set have terminated.

Siblings that are not concurrent are referred to as *serial* (or *sequential*) siblings. When discussing serial siblings, we may use the terms *later* or *earlier* (*prior*) to refer to the order in which they were created (and ran). We use the terms *serially related* and *concurrently related* for relative actions A and B to note the relation between their ancestors, A' and B', that are

[5]Think about relative actions as "cousins".

[6]The distinction is not important in the current model, which does not allow descendants of subactions in different sets of concurrent subactions (of the same parent) to be active simultaneously.

serial or concurrent siblings.

The commit of a subaction is **always** relative to its parent. When an action aborts, all the effects created by its (committed) descendants are undone. When a subaction S and all its ancestors up to the topaction committ, we say that S has *committed to the top*.

Handler Calls

An action runs at a single site only. When an action A wants to invoke a handler call to another site, another action H has to run there. This way A is isolated from failures of H's site or the network. In our model, when A at site G_A tries to use a handler on site G_H, a special subaction C called a *call-action* is created at G_A to carry out the call to G_H, where a *handler call* subaction (H) is created. The call-action C does no real work, but enables A to abort its handler call H by aborting C locally, with no need to delay until H's site is notified; such a delay can be very long when the network partitions. Note that C always commits to A, even if H aborted. For simplicity, we often ignore call-actions in the rest of this thesis when their existence has no effect on our algorithms.

Nested Topactions

Our model also includes *nested topactions*. As their name suggests, these are topactions created inside actions. Unlike a normal subaction, the commit of a nested topaction is independent of the commit of its parent (and the parent's transaction); effects created by a nested topaction become permanent upon its commit. The only difference between a nested topaction and a non-nested one is that the former was started from within some action. Nested topactions are used to perform *benevolent side-effects*, such as rearranging data objects to exhibit better performance (e.g., sort a list), without changing their observed values.

Action Identifiers

Action identifiers (*AIDs*) are used to name actions. The AID is a data structure that contains all the information describing the action's location in its action tree and its site. The AID used in this thesis is slightly different from the original one, including a change to observe the distinction between concurrent siblings and siblings of different concurrent sets (discussed above) and accommodating new features (like marking an action as disconnected). Given the AIDs of two actions of the same transaction, we can determine whether they are serially or concurrently related.

Our AID is implemented as a list of tagged numerators; a possible implementation is given in Appendix A. The list describes the action's ancestry; an action's AID is basically its parent's AID concatenated with a tagged numerator. The tag identifies the action (e.g., S for subaction, G for a new site (handler call), C for concurrent-set action and T for topaction), and the numerator differentiates among similar siblings and indicates serial ordering relation between serial siblings (except for handler calls, where the numerator indicates the site, and uniqueness and serial ordering is provided by the call-action's numerator).

Actions in examples in this thesis are given names with a format similar to their AID. For example, $G5.T84.S3.C2.T3.S1.G6$ is the name of a handler call made to site 6 by the call-action 1 of the nested topaction $G5.T84.S3.C2.T3$, which belongs to the concurrent set 2 of the third subaction of topaction 84, within site 5. For simplicity, we often use a simple short form, ignoring call-actions and new sites. For example, $A.2$ marks the second subaction of the action named A, or $A.C3.5$ marks the fifth subaction in the third concurrent set of A, created serially after $A.2$ (subactions and concurrent-set actions use the same counter to generate their numerators).

Implementation of actions

An action is implemented inside a site as a data structure (*AINFO*) that hold information about the action. This information includes the action's AID, list of objects used by the action, list of aborted descendants of the action (*alist*) and a list of sites used by the action's descendants (*plist*).

The site keeps two lists of actions, *ACTIVE* and *COMMITTED*. The *ACTIVE* list contains *AINFO*s of all local actions that are currently active. When an active action A terminates, its entry is removed from *ACTIVE*. If A aborted, its entry is discarded (and A's AID is added to its parent's *alist*). If A committed, there are three cases:

- A was a topaction: See Section 2.2.7 below.
- A was a handler call: A's entry is placed in *COMMITTED*.

- If none of the above then A must have a parent P in $ACTIVE$: A's *alist*, *plist* and the list of objects are merged into P's.

Both $ACTIVE$ and $COMMITTED$ are volatile objects. When a site crashes, the contents of both lists are lost, which has the effect of aborting the actions (that belong to active transactions) that ran at that site.

2.2.4 Atomic Objects

The use of *atomic objects* by a set of actions ensures their atomic semantics; these actions would be serializable and recoverable. While our model allows for non-atomic objects to exist[7], we restrict this thesis to atomic objects only; therefore whenever objects are mentioned, we mean atomic objects.

In our model, atomic objects synchronize the actions that access them by the use of locks. Every operation on an object is classified as a *read* or *write*, and an appropriate lock (read-lock or write-lock) must be obtained (automatically, by our system) before the operation can take place[8]. Strict two-phase locking ([Gray, *et al.* 1976]) is employed; locks are held until their action commits or aborts.

The rules for acquiring locks are derived from conflicts between operations: A write operation conflicts with any other write or read operation. Without nesting, the rules are simple: many concurrent readers are allowed, but when an action holds a write-lock on an object, no other lock can be granted on that object. With nesting, the rules are extended: An action A can acquire a read-lock if and only if all holders of write-lock are A's ancestors, and can acquire a write-lock if and only if all holders of read or write locks are A's ancestors.

We implement the locks on an object as an ordered stack of write-locks, and unordered set of read-locks[9]. The way the lock acquisition rules are applied is described in Figure 2-1 for a read-lock request, and in Figure 2-2 for a write-lock request[10]. Note that we introduce additional requirements into the current model:

---

[7] For example, non-atomic objects are needed in order to build *user-defined atomic types* (see [Weihl 1984]).

[8] Note that models exist where locking takes place on a higher level, like the type-specific locking in [Schwarz 1984].

[9] The current implementation of Argus keeps all locks together, which is a semantically confusing.

[10] Note that a can_not_have_lock reply implies that a query (explained below) has to be sent.

```
check_locks_for_read = proc(Reader:ainfo, OBJ:object) returns(reply)
    % Called at a site to check whether Reader can get a READ lock on OBJ
    if versions_stack$non_empty(OBJ.write_stack) then
        top_write_lock : lock := versions_stack$top_lock(OBJ.write_stack)
        if aid$non_descendant(Reader.aid,top_write_lock.aid) then
            return(can_not_have_lock(Reader,OBJ,top_write_lock.aid))
            end
        end
    % Reader is a descendant of the top write locker (if any)
    if lock_set$member(Reader.aid,OBJ.read_set) then
        return(already_has_lock(Reader,OBJ))
      else
        return(can_have_lock(Reader,OBJ))
        end
    end check_locks_for_read
```

Figure 2-1: The lock acquisition rules applied for a reader

- The lock acquisition rules should be satisfied on every access of an action to an object, regardless if the action already has the lock. This is a result of allowing a parent to run concurrently with its descendants in our new model, enabling descendants to "snatch" locks from objects used by their active ancestors ([Xu & Liskov 1988]).

- A read-lock is required for reading even if the action has a write-lock. The reason has to do with updating timestamps (see Section 4.2).

Atomic objects enable a site to undo modifications done by an action (when the action is aborted) by using *versions*. The state of an unlocked object is stored in a *base-version*, and new versions are kept in a stack fashion. When an action A acquires a write-lock, a new version is created for A with an initial value of the previous top version (or the base-version if no other version exists). The action modifies its version only, and if the action aborts, its version is discarded. Note that write-locks and versions are tightly coupled; hence we may use those two terms interchangeably[11].

Note again that versions on an object are kept as a stack with the following invariant holding: Actions holding versions on some object below some version V are all proper ancestors of the

[11] Write-lock and its corresponding version are also implemented as one object.

```
check_locks_for_write = proc(Writer:ainfo, OBJ:object) returns(reply)
    % Called at a site to check whether Writer can get a WRITE lock on OBJ
    for read_lock:lock in lock_set$elements(OBJ.read_set) do
        if aid$non_descendant(Writer.aid,read_lock.aid) then
            return(can_not_have_lock(Writer,OBJ,read_lock.aid))
            end
        end
    if versions_stack$non_empty(OBJ.write_stack) then
        top_write_lock : lock := versions_stack$top_lock(OBJ.write_stack)
        if aid$non_descendant(Writer.aid,top_write_lock.aid) then
            return(can_not_have_lock(Writer,OBJ,top_write_lock.aid))
          elseif Writer.aid = top_write_lock.aid then
            return(already_has_lock(Writer,OBJ))
            end
        end
    % Writer is a descendant of the top write locker (if any)
    return(can_have_lock(Writer,OBJ))
    end check_locks_for_write
```

Figure 2-2: The lock acquisition rules applied for a writer

holder of V. Actions holding versions below the top version can not access the object (they are blocked, in the current model), and the action holding the top-version (and its descendants) get the top-version's value whenever they read the object.

When a subaction commits, its locks and versions are propagated to its parent. The commit of a topaction is handled differently as described in Section 2.2.7 below. Propagation means that the action's locks and versions become the parent's, and if the parent had locks or versions (on the same objects) before, they are replaced with the new ones. All the versions in the stack above the parent's new version are discarded (so the parent's version becomes the top one). When a subaction (or topaction) aborts, its locks and versions are discarded.

While base-versions are kept in stable storage, locks and versions are volatile. Therefore when a site crashes, not only are all the actions that exist there (of active transactions) aborted, but all their locks and versions are lost.

Lock and version propagation is done in a "lazy" fashion when the parent of the committing action lives on another site. Locks and versions of a handler call H are not modified to become the parent's when H commits. Also when an action is aborted, abort messages are not guaranteed to arrive at all the sites used by the action's descendants; therefore some objects may appear locally to be locked even though their current holder has aborted. Only when another action A tries to acquire a lock on some object O, locked in conflicting mode by H (e.g., A tries to write O, which is read-locked by H), then a series of queries is initiated in an effort to propagate the lock up the action tree to the LCA of A and H. Once the lock becomes the LCA's, A can acquire it.

Queries are implemented as follows: The object O's site (G_O) sends repeated messages to the site of the LCA of A and H (G_{LCA}). G_{LCA} can tell (by examining G_{LCA}'s *ACTIVE* and *COMMITTED* lists) whether:

1. H has committed up to the LCA.

2. Some ancestor of H has aborted.

3. Some ancestor of H is (possibly) still active elsewhere.

In the first two cases, G_O can do the local processing needed to grant a lock on O to A (provided no other conflicting locks exist on O). In the third case, G_O has to repeat querying G_{LCA}. A possible optimization in this case is to query sites of ancestors of H that are descendants of the LCA; if one of these aborts then G_O needs not wait for a reply from G_{LCA}.

2.2.5 Graphic Representation

We often use graphic representations of action trees in examples given in this thesis. Figure 2-3 demonstrates such an action tree. The following conventions are used:

- Ovals represent sites (e.g., G_A).

- Ellipsis represent actions (e.g., $A.1$).

- Rectangular stacks represent objects (e.g., O_1) with their versions (e.g., $A.1$'s version).

- Direct arrows mark the relation parent $\longrightarrow$ child.

- A curved arrow marks serial order between sibling subactions.

- A pair of parallel curves marks concurrent sibling subactions.

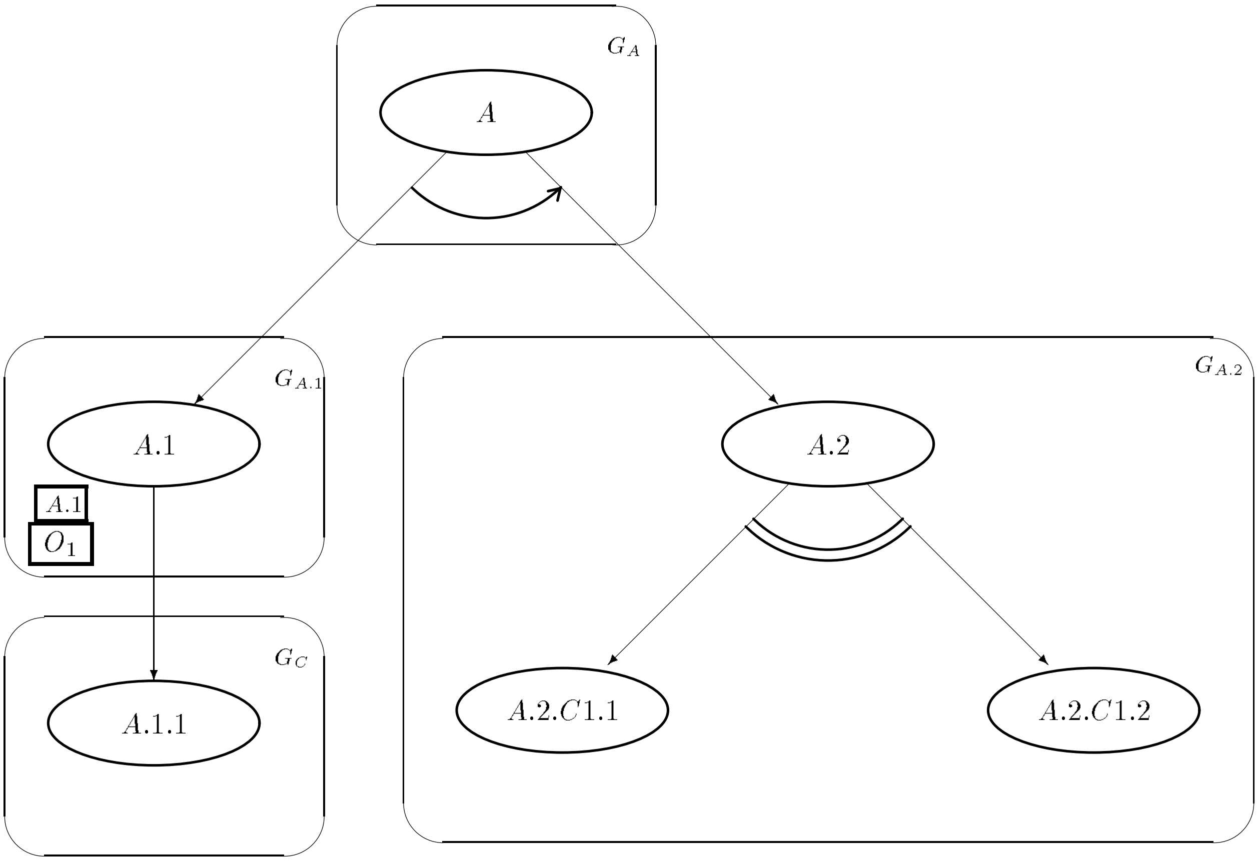


Figure 2-3: An example for an action tree.

- Absence of any of the two previous marks means that the relation between the siblings is not specified.

For simplicity, we often ignore some irrelevant details in our pictures. For example, call-actions are often ignored and sometimes guardians or objects are not drawn when we focus on things like the relation (serial or concurrent) between actions in the action tree.

2.2.6 Orphans

An *orphan* is a computation that is active even though its results are no longer needed. For example, suppose an action (e.g., A in Figure 2-3) aborts its handler call (e.g., $A.2$) due to a network partition. The handler call and its descendants may still be active after being aborted since the abort message has not arrived at their site ($G_{A.2}$ in our example). We call such an action ($A.2$) or any of its descendants *abort orphan*, since it is a result of an abort.

Another kind of orphan is the *crash orphan*. A crash orphan is an action that depends on the volatile state of a site that was lost when that site crashed. For example, suppose that $A.1$ (in Figure 2-3) created a handler call $A.1.1$, within the site G_C and committed back to $A.1$. If G_C crashes while $A.1$ is still active, then $A.1$'s results would become invalid[12]. If G_C crashes after $A.1$ had committed, but with A still not committed, then A and all its descendants are considered crash orphans.

Orphans are undesirable because they waste resources (taking processor time and preventing other actions from using objects locked by the orphans) and may observe inconsistent information, possibly resulting in unpredictable behavior.

Algorithms to detect both kinds of orphans ([Liskov, *et al.* 1987a]) exist in the current model. The method to detect abort orphans works by keeping track of all aborted actions that may had active descendants. It is implemented by keeping the AIDs of aborted actions in a data structure called *done* at each site G (as $G.done$). Updates (i.e., newly aborted actions) are propagated to other sites by piggybacking the local *done* on (almost) every message M as $M.done$. Sites abort all descendants of actions in their *done*. A proof exists that the algorithm to detect abort orphans prevents these orphans from seeing inconsistent states ([Herlihy, *et al.* 1987]).

Crash orphans are detected by maintaining, at each site, a *crashcounter* that is incremented each time a site recovered from a crash, and a *map* that reflects the site's current knowledge about the values of the *crashcount*s of other sites. Each action A maintains a data structure called the *dlist*, which contains the names of all the sites on which A *depends*, i.e., whose crash would make A a crash orphan. The algorithm works by having messages propagate information ($M.map$) to sites, and "catching" orphans, when a message M arrives at a site G, in one of two ways:

- If M is a call or reply, and its *dlist* contains a site C such that $G.map(C) > M.map(C)$, then if the message is a call, the caller is an orphan (and no handler action is created), else the action replied to is an orphan (since it depends on its handler call) and is aborted.

- If some site C has to be updated in $G.map$ (i.e., $G.map(C) < M.map(C)$), then any local action that depends on C is aborted.

---

[12] More precisely, this requires $A.1.1$ to have locks in G_C or to have made handler calls to other sites (and acquire locks there) and so on before G_C crashed, but this is most often the case.

Actions' *dlists* are managed as follows: A topaction's *dlist* initially contains the local site only. A subaction's *dlist* is initially its parent's *dlist* plus its own site. When a subaction aborts, its *dlist* is discarded; when it commits, its *dlist* is merged (as mathematical set union) into its parent's *dlist*. Finally, when a descendant inherits a lock from an ancestor, the ancestor's *dlist* (as of the time the ancestor acquired the lock) is merged into the descendant's.

The reason for adding ancestors' *dlists* to the *dlist* of the action that inherited a lock is concurrency. Without concurrency, only a single point is active in the action tree at any time (before the topaction commits), and this point traverses the tree in a depth first order, so passing the *dlist* at creation and commit times is enough to ensure that an active action always has a correct *dlist*. With concurrency, a concurrent subaction S_2 may acquire a lock on an object that was modified by a concurrent sibling S_1, thus making S_2 depend on S_1's commit, and requiring adding S_1's *dlist* to S_2's. The implementation simplifies the process by taking, instead of S_1's *dlist*, the *dlist* of $LCA(S_1, S_2)$, which includes the *dlist* of S_1 plus possibly *dlists* of other committed concurrent siblings. The implementation is as follows[13]: When a site propagates write-locks of some subaction S to some ancestor P of S, P's current *dlist* is associated with each propagated version. When P is not local, a query is sent and the reply "S committed up to P" must be received before that propagation happens. That reply message contains P's current *dlist*. Independently from the query and propagation process, when an action acquires a lock on an object that was modified by its transaction, all the *dlists* associated with the versions on that object must be merged into the action's *dlist*. Note that associating *dlists* with versions allows us to remove *dlists* when actions abort and their versions are discarded.

2.2.7 Committing Transactions

Topactions commit in our model by executing a top-level commit protocol known as *Two Phase Commit* ([Gray 1978]). This protocol ensures that the transaction either commits everywhere or aborts everywhere. The participants in the protocol are the sites in the committing topaction's *plist* (see Section 2.2.3 above); the coordinator is the topaction's site[14].

[13]This process is not clearly defined in [Walker 1984, Liskov, *et al.* 1987a, Nguyen 1988]. The description here is our understanding of the way it should be done.

[14]Note that the topaction's site is also a participant.

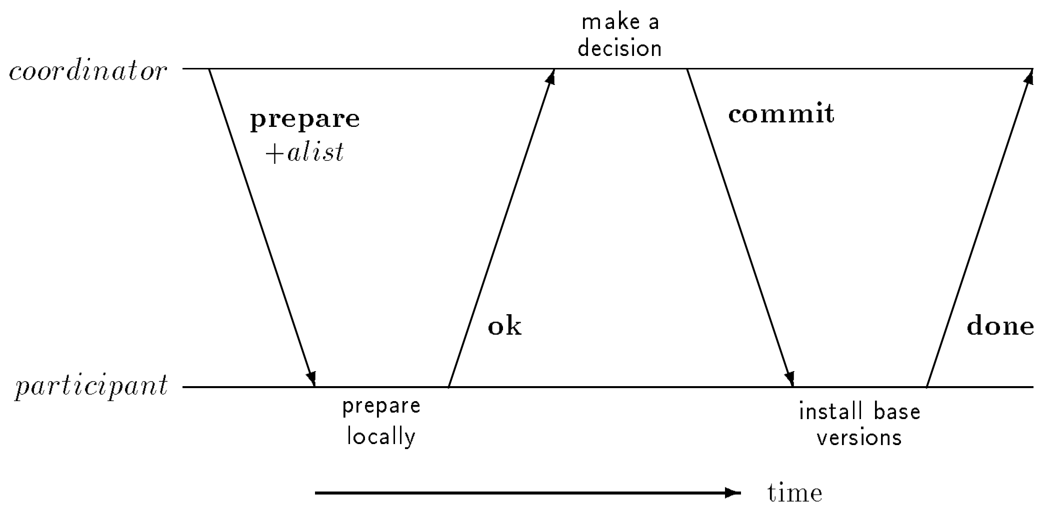


Figure 2-4: A time-space diagram of the current Two Phase Commit protocol.

The protocol is depicted in the time-space diagram in Figure 2-4; time elapses from left to right, and separate sites have separate horizontal lines; the slanted arrows show messages sent between the coordinator and a participant. Though only one participant is shown in the diagram, all the participants receive and send similar messages in the same time interval. Note that our diagram shows only the case where the protocol succeeds; the case where the transaction is aborted is explained below.

The coordinator begins the first phase by sending a *PREPARE* message to all the participants. The message is accompanied by the committing topaction's *alist*. Each participant that received the *PREPARE* message prepares locally by recording on stable storage a tentative version for every object that was modified by the transaction. The participant is considered prepared after all the versions have been recorded and a *PREPARE* record have been written to stable storage, since this allows it to recover from a subsequent crash and continue participating.

Since our model propagates locks in a lazy fashion, a preparing participant may find several versions stacked on an object to be prepared. The participant can determine the correct version to record by using the *alist* that came with the *PREPARE* message. The correct version to prepare is the top-most version whose owning action A is not in the *alist* (i.e., A committed to the top).

The participant that prepared successfully replies with *OK* to the coordinator; a participant that is unable to prepare (e.g., because it has crashed and lost all the relevant information) replies with *REFUSED*. After replying, the (prepared) participant can release all the read-locks held by the transaction.

The coordinator decides on the transaction's fate based on the replies from the participants. If all replied with *OK*, the coordinator commits the transaction by writing a *COMMITTED* record to its stable storage. If some participant responded with *REFUSED*, or did not respond within a predetermined period (timeout), the coordinator has to abort the transaction.

In the second phase, the coordinator notifies the participants of its decision. If it decided to commit, it sends a *COMMIT* message to the participants, which record the commit on stable storage, replace the base versions of the transaction's objects with the tentative versions, release the remaining locks of the transaction and reply with *DONE*. If the coordinator decides to abort the transaction, it sends an *ABORT* message, and the participants discard all the transaction's locks and tentative versions.

A participant P that did not receive the coordinator's decision (e.g., because P crashed before the second phase) may query the coordinator later for its decision, which must be kept by the coordinator's site (unless all the participants replied with *DONE*).

Chapter 3

Disconnected Actions

The purpose of this thesis is to extend our *current* model of computation for a nested atomic action system, as presented in Chapter 2, to accommodate disconnected actions. This chapter describes how DAs fit into the model, as opposed to *regular* (i.e. not disconnected) subactions.

This chapter is organized as follows: We begin by describing DAs and the role that they fill in a nested action system and their semantics. Next we describe the design, terminology and implementation of DAs in our model. And the last two sections describe the necessary changes to the query and orphan detection mechanisms to support DAs.

3.1 What are disconnected actions?

Nested actions enable programs that use atomic actions to exploit parallelism by providing a tool to performs several tasks concurrently within an action. However, there are cases where nested actions can not take advantage of potential parallelism, and independent tasks have to be done sequentially. For example, many abstract data objects provide operations that perform some abstract modification to the object; the implementation of these operations often needs to do more work internally on the object's representation to save some work for the subsequent operations. The additional improvement work can be done in parallel with the work that the caller (i.e., the action that called the operation) does after the operation. Doing these two tasks in parallel is not possible in our current model of nested actions.

Take the *Fibonacci Heap* ([Fredman & Tarjan 1984]) as an example for an abstract data object. With n elements in the heap, the common operation of *extract_minimum* takes a time of $O(\lg n)$, while *read_minimum* is brief ($O(1)$). The difference in time results from the work that has to be done to rearrange the heap after extraction of the minimum. A program that has to extract the minimum can be speeded-up by reading the minimum first (and marking it as obsolete), and doing the extraction work "in the background", in parallel with the continuing program. To use such a speed-up technique in the current model, the program needs to do the following:

```
begin topaction % A
    begin action % A.1
        min := bonacci_heap$read_min_and_mark_it()
        coenter % create a set of concurrent subactions
            action bonacci_heap$nish_extracting_minimum()
            action do_rest_of_work % of A.1
            end
        end % A.1
    do_rest % of A
    end % A
```

The above method has the following drawbacks: Violation of abstraction (the caller needs to know about the two operations), awkward programming (squeezing the real body of the action $A.1$ into a concurrent subaction), and limited scope (the operation nish_extracting_minimum can not run in parallel with the rest of the transaction, i.e., A).

Cases that require "background" work, as described above, can be better handled when the current model is extended to have *Disconnected Actions* (DAs). A DA is a subaction that executes in parallel with the rest of the transaction, and performs benevolent side-effects within the parent action. An action A that creates a DA D need not be blocked until D terminates (as is the case with creating regular subactions); A continues to run in parallel with its disconnected child D. The disconnected action D is executed as a subaction, but asynchronously with its transaction; that is, its creator A can run and commit (assuming A is not a topaction) to A's creator, and so on, without any effect on D. The only event that synchronizes D with its transaction is the termination of D's topaction; no (non-orphan) DA is left active after its topaction terminates.

With the use of DAs, the previous example can result in a cleaner program and the implementation of the abstract operation can hide the "background" work:

```
extract_min = handler () returns (element)
    min : element := bonacci_heap$read_min_and_mark_it()
    disconnected_action bonacci_heap$nish_extracting_minimum()
    return (min)
    end
```

and the program can use the operation in a simple manner:

```
begin topaction % A
    begin action % A.1
        min := bonacci_heap$extract_min()
        do_rest_of_work % of A.1
        end % A.1
    do_rest % of A. (Note that the DA may still be active!)
    end % A (including the DA)
```

3.1.1 Kinds of Disconnected actions

A disconnected action can be used by a programmer as a single operation that creates some benevolent side-effects (like improving the representation of an object, as described above).

Another use of DAs by programmers is for *quorum-sets* that perform successfully at least a quorum M of operations out of a set of N (such that $M \leq N$). For example, quorum-sets can be used to update replicated objects, as described in Section 1.1. The programmer has to use a statement like:

$$\texttt{do } M \texttt{ of } \{OP_1, OP_2, \ldots, OP_N\}$$

and handle an exception when fewer than M actions (operations) can commit.

3.1.2 Semantics of Disconnected Actions

The commit of a disconnected action, like that of any subaction, is relative to its parent; if the parent (or any of its ancestors) aborts, all the effects done by the DA are undone.

The DA is terminated by the time its topaction terminates. Since DAs run asynchronously with the rest of the transaction, the commit protocol of the topaction must be modified to ensure termination of DAs of that transaction. These changes are discussed in Chapter 5.

The fate (i.e., commit or abort) of a DA does not effect the commit decision of any of its proper ancestors (except the case of quorum-set DAs discussed below). Therefore, DAs can not be used for doing "real" work, only for benevolent side-effects.

Disconnected actions must have the same termination semantics as regular subactions; a disconnected action must appear, to the rest of the transaction, to be terminated as soon as its creator resumes its execution. Since DAs are active after their creator continues (unlike regular subactions), changes must be made to the current model to ensure serialization of DAs; these changes are discussed in the Chapter 4.

3.2 Adding Disconnected Actions to the Current Model

This section describes issues related to design, terminology and implementation of our system model with disconnected actions.

A goal in the design of the new model was to minimize the change to the design of the current model and its performance. This goal was achieved; when DAs are not used, the new model does not send any extra messages, does very little additional local processing, and requires little additional storage.

The term *top DA* is used for a disconnected action that was created by a regular action[1]. A top DA can run and create subactions like any other action. The term *disconnected action (DA)* is used in this thesis to refer to any of the top DA or its descendants.

Our model of disconnected actions can be generalized to have *nested disconnected actions*; that is, disconnected actions created by disconnected actions. Allowing disconnected actions to be created recursively can be useful, for example, when DAs are used to implement abstract services, which in turn may be used by DAs.

Most of the work in this thesis is concerned with non-nested DAs. Nested DAs are discussed in various level of detail; in most cases they seem to require only a straightforward extension of the work for non-nested DAs.

All the quorum operations are run as DAs[2]. The creator of the set of (N) concurrent operations is blocked until at least a quorum of (M) disconnected subactions commit. This way there is no need to specify a priori which operations are in the quorum, and the rest get an

[1]The term *top DA* is also used for a nested DA created by a DA and so on.

[2]Unlike the simplistic description in Section 1.1, where the quorum was achieved by regular subactions first, and DAs were then made to finish up.

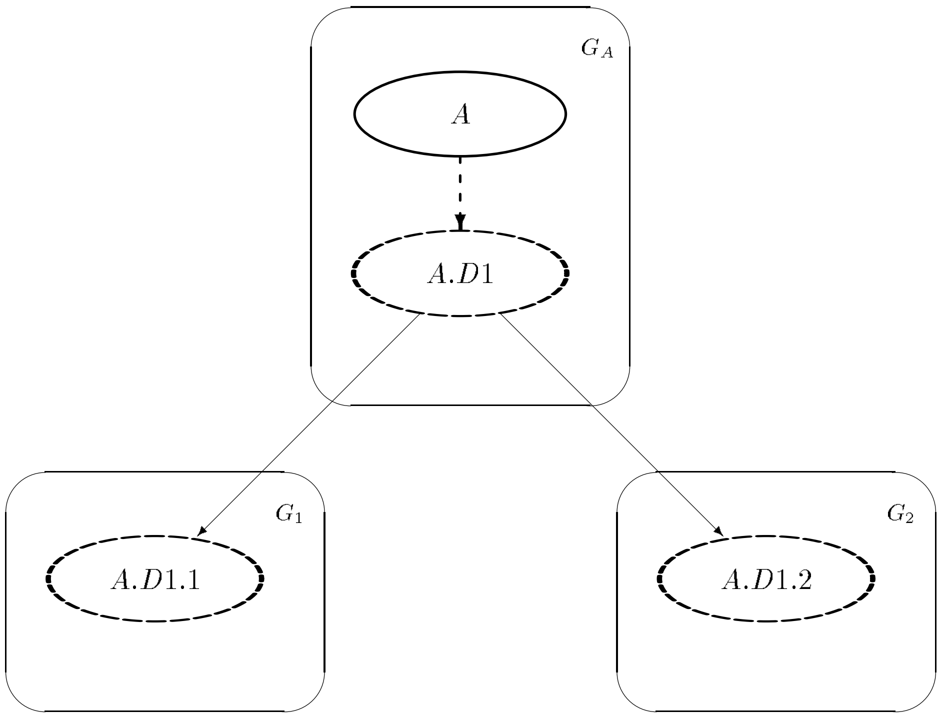


Figure 3-1: An example of a Disconnected Action

early start. Note that since the creator is blocked while those DAs in the quorum are active, their chance of success is equivalent to that of regular subactions as they do not conflict with later activity (see Chapter 4).

3.2.1 Graphic Representation of DAs

Figure 3-1 shows an example of an action A that created a disconnected action $A.D1$. We use a dashed oval to represent a disconnected action and a dashed arrow to mark the relation parent_action $- - \rightarrow$ child_top_DA. The DA ($A.D1$) can create subactions like any other action, therefore we use the continuous arrows to mark relations between a disconnected action and its children (e.g. between $A.D1$ and $A.D1.1$). Dashed arrows are used only for marking the "disconnected branch" in the action tree. Though the "disconnection" always happens inside a site, we may simplify some descriptions and draw dashed arrows across site boundaries (e.g., in Figure 6-1).

3.2.2 Implementation of Disconnected Actions

The top disconnected action is implemented as two actions, a parent and a child. The parent does no real work (similar to a call-action or a concurrent-set action); it only creates a child and commits after the child terminated. The child does the work like any regular subaction. The reason for breaking the top DA in two is that some record must be kept at the site when the top DA aborts; since the parent always commits, it will be entered in *COMMITTED*, and its child will be in its *alist* if the child aborts.

Putting the DA's entry in *COMMITTED* provides a simple way to keep separately information about that DA, including its *plist* (needed by the algorithm in Chapter 6), its *dlist* (needed by the modified orphan detection scheme in Section 3.4) and its *alist* (needed by the modified Two Phase Commit protocol in Chapter 5).

Concurrent-set actions, of only explanatory value in the current model, need to be implemented in the model with DAs. They should be stored in *COMMITTED* after committing when they are used for quorum sets because they maintain data structures that are used by active quorum DAs and needed for our fault-tolerant commit method (described in Chapter 6).

Like call-actions, concurrent-set actions and the distinction between parent and child for top DAs are often ignored in our examples and discussions.

Quorum sets are implemented as follows: The concurrent-set action C, which creates a set of concurrent quorum DAs, maintains a data structure to monitor the current state of its quorum DAs. C's parent blocks when C is created, then C creates its concurrent set of N DAs and blocks until a minimal quorum of Q DAs has committed, at which point C commits and its parent continues. If a quorum can not be achieved (i.e., at least $N - M + 1$ DAs aborted), C aborts. Whether C is active or committed, its site keeps monitoring the state of its DAs until they all terminate; this information, kept with C's entry (in *COMMITTED*), is needed later by the Fault Tolerant Two Phase Commit method (see Chapter 6).

3.3 Queries

To use disconnected actions, the current lock query mechanism must be changed. In the current model, when some action S needs a lock on an object that is locked in a conflicting mode by an action L, queries must be sent to the site of $LCA(S, L)$ to find out if L committed all the way to the LCA. Queries can be sent to sites of ancestors of L (that are descendants of the LCA) as an optimization to detect a possible abort of an ancestor of L earlier.

When a lock holder is a disconnected action, queries to the LCA are not sufficient to determine that the lock holder committed to the LCA. An ancestor P, disconnected from the locker L, may commit to the LCA while L is still active. Only when the top DA ancestor of L commits (and P also commits to the $LCA(S, L)$) can L's site propagate L's locks and versions to $LCA(S, L)$. To generalize for the case of nested DAs, lock propagation from a DA L to an ancestor A requires that all top DAs between L and the A commit, in addition to the current requirement that an ancestor of L that is a child of A must commit. As before, if any ancestor of L was found to be aborted, L's locks and versions are discarded.

To grant a lock on an object locked (in conflicting mode) by a DA D, queries must be sent to the LCA and to all the sites where top DAs, ancestors of D that are descendants of the LCA, were created. Figure 3-2 depicts an example of an action $A.2$ that tries to get a lock on the object O that is (write) locked by the nested DA $G1.A.G2.G3.D1.G4.G5.D1.G6.GO$. The thick arrows mark the queries that G_O must send. G_O deduces from the locker's AID that it has to send queries to G_1, the site of A (LCA of $A.2$ and the locker), to G_3, the site of the top DA $G1.A.G2.G3.D1$ and to G_5, the site of the nested top DA $G1.A.G2.G3.D1.G4.G5.D1$. All the queried sites must reply that the relevant action committed before G_O can grant the lock (and the top version) on O to $A.2$; if some site informs that the relevant action aborted, the locker's locks and versions should be discarded.

The implementation of the modified query mechanism uses the locker's AID to deduce the list of sites and ancestors (of the locker) for the queries (the needed operation, "locations_of_disconnected_ancestors", is described in Appendix A). The site that initiates the queries traverses the AID of the disconnected locker in a bottom-up order, propagating the ownership of locks and versions up the tree; only when a reply contains information about the commit of a top DA D can the the ownership pass to D's parent.

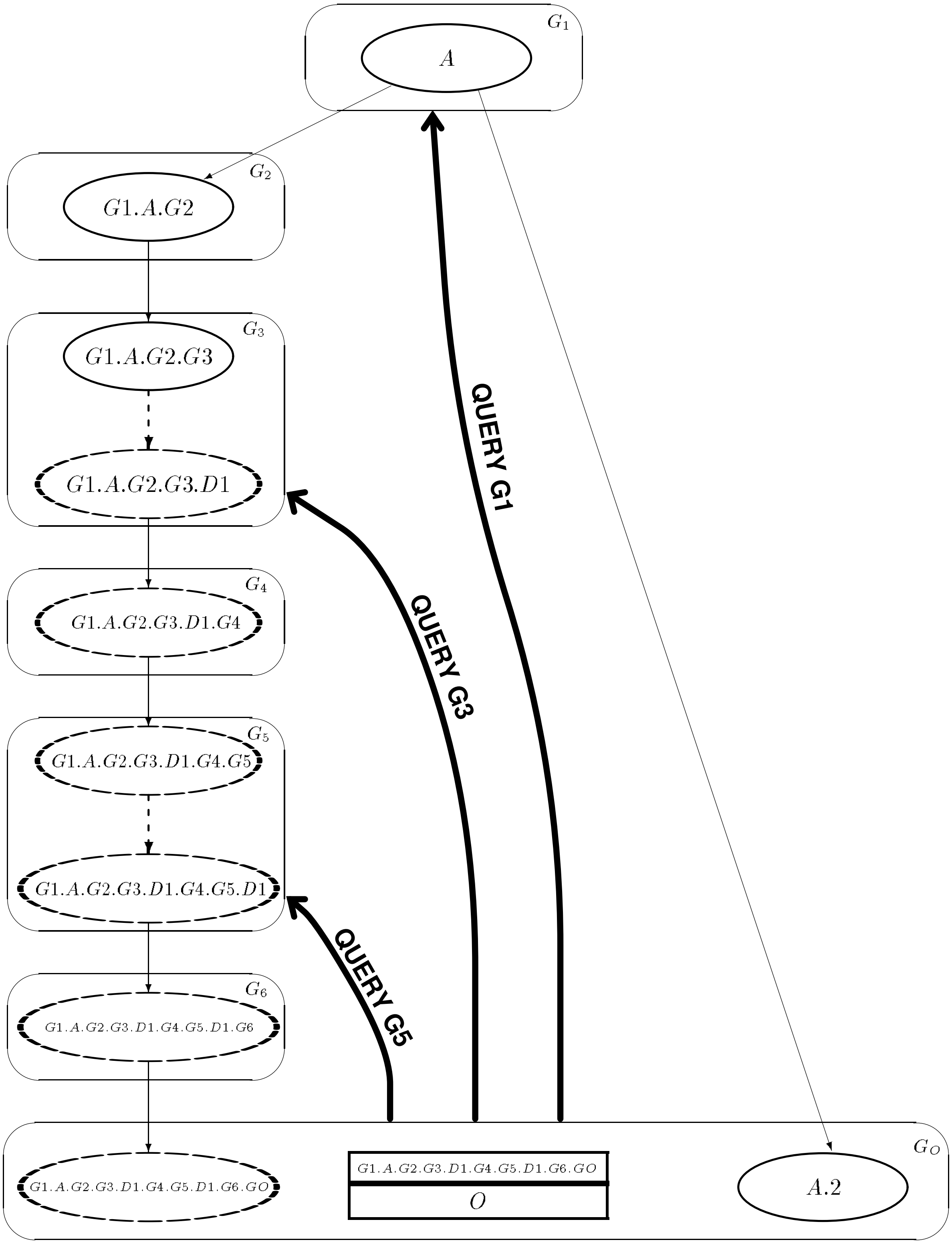


Figure 3-2: An example for a site making queries about the fate of a lock owner.

3.4 Interaction with the Orphan Detection mechanism

The orphan detection algorithms are distributed algorithms that propagate knowledge among sites piggybacked on messages sent by the transactions. The crash orphan detection algorithm also associates information with the actions themselves.

The only difference between the system model that uses DAs and the current one, from orphan detection point of view, is that some information is not passed from a committing top DA to its parent, unlike the case between any other action and its parent.

The abort orphan detection algorithm associates no knowledge with specific actions and does not pass any information from a committed action to its parent. Therefore this algorithm should work in a system model with DAs as before.

The crash orphan detection algorithm propagates knowledge among sites in a manner similar to the algorithm for abort orphans, but in addition associates *dlist*s with actions. Updates to *dlist*s travel along branches of the action tree (with subaction's creation and commit) and with ("committed") replies to lock queries to the LCA.

In a model with DAs, the *dlist* of a top DA D is not merged into the *dlist* of its parent when D commits (because the parent, and several other ancestors, may have committed by that time). This poses no problem for an ancestor P of a DA D that does not belong to a quorum, because P can commit even if some participant of D crashes, as explained in Chapter 6. However, when some relative R of D observes D's effects, it becomes dependent on D's commit (and *dlist*). If R is a regular subaction and later commits to P, P would also depend then on D's *dlist*.

An action observing effects of a relative DA can get the DA's *dlist* using the extended query mechanism (Section 3.3). Basically replies to queries sent to sites of top DA should also include the committed top DA's *dlist*. As can be seen in the example of Figure 3-2, the site with the object (G_O) sends queries to G_3 and G_5, in addition to the LCA's site G_1. The combination of the *dlist*s of the three replies would cover the *dlist* of the version holder that committed to the LCA.

A problem does exist with DAs that are used for quorum sets. An ancestor of a quorum set depends on the commit of at least a quorum of its descendant DAs. Three solutions are proposed here:

- Do nothing. Some crash orphans may not be detected.
- Pass the *dlist*s of the DAs that committed as part of the quorum to the concurrent-set action. Some non-orphans may be detected as crash orphans.
- Pass more information with the *dlist*s of committed DAs to the concurrent-set action. When an action is detected as an orphan because some site used by a DA of the quorum crashed, the site that created the concurrent-set action must be queried to know if the quorum minimum was violated.

The last solution is too complex, and requires a change to the orphan detection algorithm. With the use of the fault tolerant commit algorithm discussed in Chapter 6, quorums sets done using DAs are likely to survive crashes, so the first solution is favored; without that algorithm, the second solution can be used.

Chapter 4

Serialization of Disconnected Actions

In this chapter we describe the way our new model, which supports disconnected actions, is extended to ensure serialization of atomic actions. In the current model, actions were serialized by obeying a strict two-phase locking protocol; disconnected actions, which run asynchronously with their parents, may violate that protocol. This chapter presents additional rules and mechanisms that enforce serial behavior in our new model.

One atomic action is serialized after another if they access atomic objects in conflicting mode; that is, at least one of the two updates the object. In this chapter, whenever we talk about an access to an object, we mean access in conflicting mode.

An important point in our serialization mechanism is the ordering relation between the two relative actions (i.e., of the same transaction) that access an atomic object, the one that got a lock on the object and the other that tries to get a conflicting lock. The two can be serially related, which means that their LCA created their ancestors to run in a predetermined order, or concurrent relatives that run concurrently and get serialized by the use of atomic objects. We distinguish between the case of "serial DAs", which are DAs related serially to other actions, and "concurrent DAs", which are concurrently related to other actions, as we describe our solutions that handle the two cases separately .

We devised two serialization mechanisms, one for each ordering relation. In the serial case, where predetermined information about the order of the related action exists, we use a simple *timestamp* mechanism to enforce this order. For the concurrent case, where the ordering is set dynamically in various locations, we use a mechanism of *constraint tables* to centrally monitor the ordering and prevent disconnected actions from violating this order. The constraint tables mechanism also requires introduction of a new kind of lock, the *fingerprint lock*, to maintain information about access history to objects.

We give only intuitive reasoning for the correct functionality of our new rules and mechanisms; no formal correctness proof is given. Our belief in the correctness of the new mechanisms is also supported by a successful simulation we did of the new model.

The chapter is organized as follows: Section 4.1 presents an example for a non-serializable execution as a result of using DAs. Section 4.2 handles only the case of serial DAs by introducing the timestamp mechanism and additional lock acquisition rules that use timestamps. Section 4.3 completes the work by introducing the constraint tables mechanism to handle concurrent DAs and presenting the full scheme for lock propagation and acquisition in our new model. We conclude in Section 4.4.

4.1 The serialization problem

The following simplified example demonstrates the serialization consistency problem. Suppose we have a replicated counter C, composed of three replicas C_1, C_2 and C_3 (represented as $(Value_1, Value_2, Value_3)$). Read and write operations are performed on the counter using a simple majority scheme, similar to our initial example (Section 1.1): Read from two replicas; write to two replicas. The write operation is augmented by a disconnected action that updates the third replica.

Figure 4-1 shows the action tree[1] of an action A that tries to increment the replicated counter C twice. For each increment operation A creates a subaction that reads the counter, calculates the new value, writes the new value to two replicas and leaves a DA behind to write to the third replica after the subaction commits.

Note that the ordering relation between the two increment subactions, $A.1$ and $A.2$, was

[1] Sites, call-actions, etc., are ignored for simplicity.

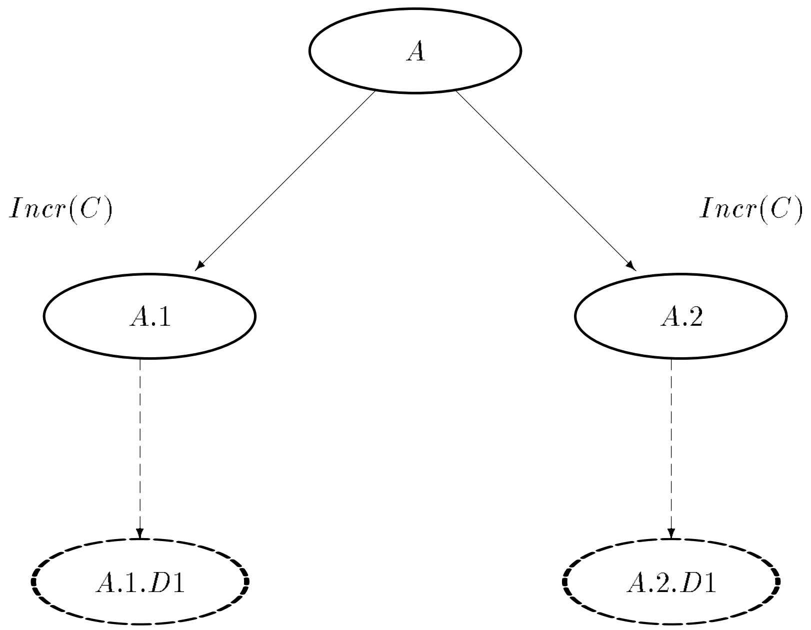


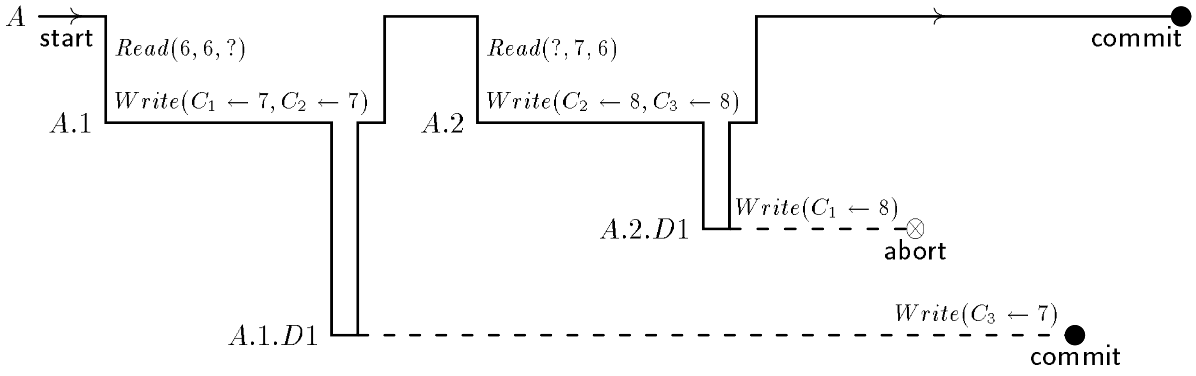
Figure 4-1: An action tree for the action A that updated the counter C twice.

not specified. The non-serializable scenario, describe below, can take place when the two subactions are either concurrent or serial siblings. The relationship between the siblings becomes important, however, for the design of the solution.

The non-serializable execution is shown in the time-space diagram in Figure 4-2. The thick line is the thread of execution, with the time passing from left to right. The dashed line marks execution of DAs. The vertical axis shows separate locations (i.e., actions). Read and write operations are given above the thread line. The states of the counter C are shown on the bottom of the figure.

The execution is as follows: The counter's value is $(6, 6, 6)$ when A begins and creates $A.1$[2]. $A.1$ reads the value 6 from the counter, writes 7 to replicas C_1 and C_2, creates a DA ($A.1.D1$) to write 7 to C_3 and commits to A. Next $A.2$ reads the value 7 from the counter, writes 8 to replicas C_2 and C_3, creates a DA ($A.2.D1$) to write 8 to C_1 and commits to A. Next the latter DA, $A.2.D1$, aborts and does not affect C_1. The first DA, $A.1.D1$, runs in parallel with the rest of the transaction and manages to write 7 to replica C_3 and commit just before A commits.

[2]Though both the thread graph and the AIDs show $A.1$ and $A.2$ as serial siblings, the same execution can take place with the two running concurrently.



$$C : \ (6,6,6) \ \rightarrow \quad (7,7,6) \ \rightarrow \qquad\qquad (7,8,8) \ \rightarrow \qquad\qquad (7,8,7)$$

Figure 4-2: The thread of execution of the action A with disconnected actions that led to an inconsistent state of the counter C

The result is a non-serializable execution! Starting with the consistent state $C = (6, 6, 6)$, the action A performed successfully two increment operations and yielded an inconsistent state. Actions following A may read the counter as either $(7, ?, 7) = 7$ or $(7, 8, ?) = 8$. The current lock inheritance and acquisition rules were not violated: A inherited the locks of its committing descendants, and each of them acquired locks that were either free or held by A; yet the execution of the two subactions is not serializable because no order exists such that performing the two subactions serially in that order would produce the same effects.

The problem was caused by the $Write(C_3 \leftarrow 7)$ operation of the disconnected action $A.1.D1$. Because it wrote to the replica C_3 after $A.2$ has also written to it, $A.1.D1$ and all its ancestors up to the least common ancestor A (i.e., $A.1$) ought to come after $A.2$ in any equivalent serial execution, while the writing order to C_2 in our example forces the opposite order. The DA $A.1.D1$ violated the basic rule of two-phase locking; it acquired a lock (on C_3) after its ancestor begun releasing locks (on C_2).

The problem could have been solved if knowledge about the serialization order existed at the site of C_3 to prevent $A.1.D1$ from writing to C_3 after $A.2$ did so. In this particular example such knowledge did exist, but at the application level, in the form of replica version numbers;

replica C_3 with value 8 has a higher version number than the one to be written by $A.1.D1$. We would like to have a general mechanism to provide similar knowledge at the system level.

4.2 Using Timestamps to Serialize Serial DAs

The serialization problem can be solved for lock conflicts between serially related (disconnected) actions with the use of timestamps. The timestamp mechanism proposed below enforces creation order on DAs; therefore it can have no effect when the conflict is between concurrent (disconnected) relatives. Section 4.3 proposes another mechanism, constraint tables, that handles concurrent DAs. Though constraint tables can be extended to handle conflicts between serial relatives as well, their implementation is expensive. The timestamp mechanism is very efficient (e.g., no extra messages needed) and handles most of the lock conflicts; therefore both mechanisms are implemented, and the decision about granting a lock to a disconnected action is made based on the ordering relation between that DA and the owner of the conflicting lock.

The timestamp mechanism proposed here enables a disconnected action D to tell that an object O was used by some future action (e.g., later serial sibling), thus preventing D from acquiring a conflicting lock on O. The use of timestamps divides the execution into intervals of pseudo-time; a new interval begins immediately after an action A creates a (top) disconnected action D (or a concurrent set of DAs), before A continues. Everything that A and its ancestors do after D's creation belong to later intervals of pseudo-time, and D can only observe effects that were created in its interval or previous ones, and change them if no other later action has yet observed these effects.

The implementation maintains a timestamp with every action and every lock. Actions' timestamps are increased monotonically during the execution of the transaction; locks' timestamps are updated when actions access objects. Thus enough information exists at a site to decide whether a DA requesting a lock can get it or must abort.

The use of timestamps can solve the serialization problem of Section 4.1 for serial siblings. Figure 4-3 shows again the action tree for A that created serial subactions. The actions' timestamps are shown (boxed) for an execution that traverses the tree in a depth-first, left to right manner. A starts with a zero timestamp and creates $A.1$ with the same timestamp. $A.1$ creates a DA $A.1.D1$, which is also given its parents' timestamp, and $A.1$'s timestamp is

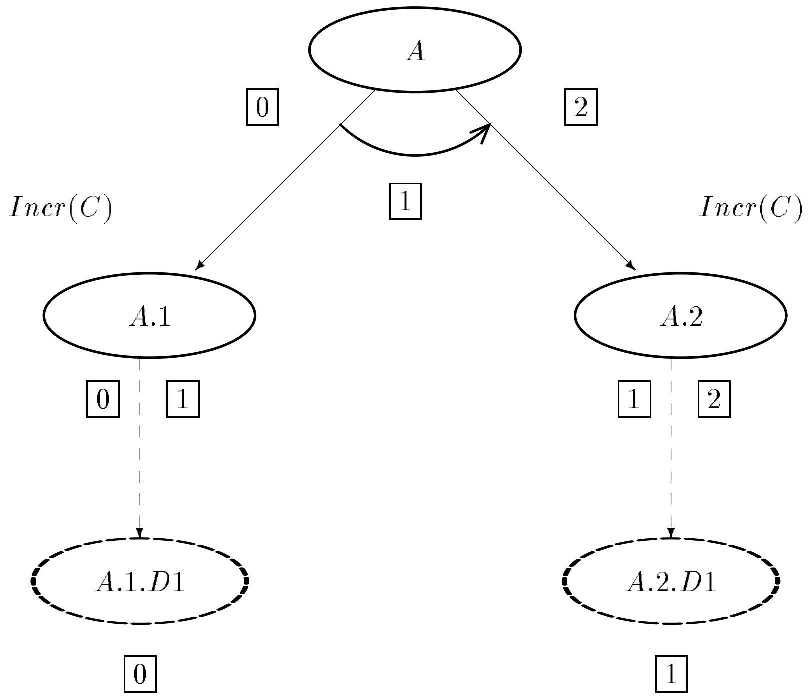


Figure 4-3: An action tree, with timestamps, for the action A that updated the counter C twice using serial subactions.

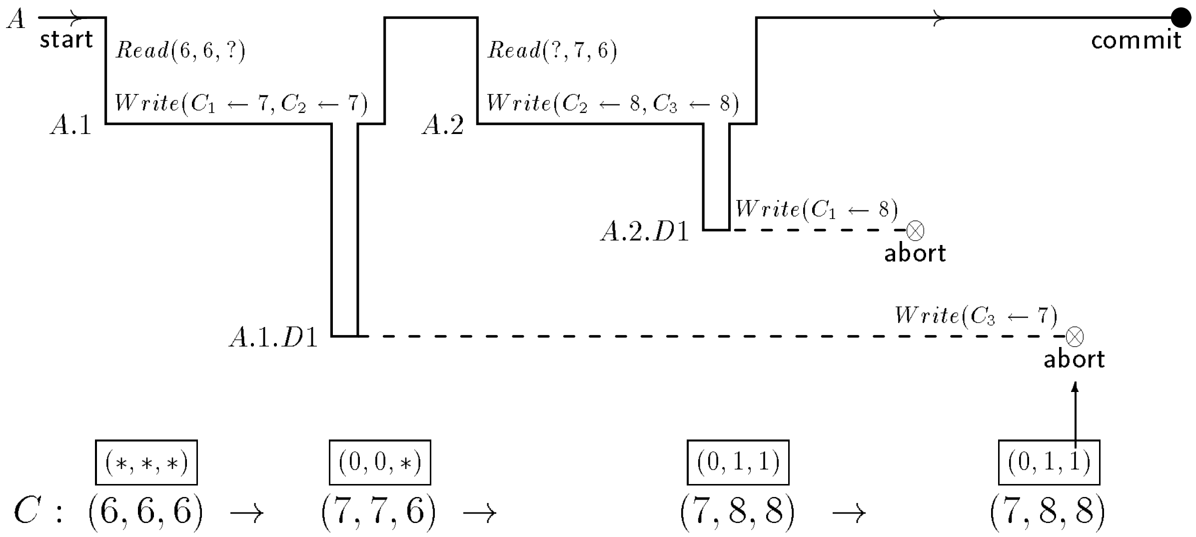


Figure 4-4: The thread of execution of the action A with disconnected actions that uses timestamps to maintain a consistent state of the counter C

incremented to 1 before it continues. When $A.1$ commits to A, A's timestamp is updated to match $A.1$'s, and a similar scenario takes place for $A.2$, which commits to A with a timestamp of 2.

Figure 4-4 shows the thread of execution and the states of the object, with their corresponding timestamps ($*$ marks an initial value). As seen, every modified replica has the timestamp that the action that modified it had at the time of the modification. When the DA $A.1.D1$ tries to modify C_3, the timestamp there has the value 1 reflecting $A.2$'s access to C_3. Since $A.1.D1$'s timestamp is only 0, it should not be allowed to read or write C_3, which has a later value, and must be aborted.

This rest of this section describes first how new locking rules must be added to the current ones to ensure serialization of serial DAs by using the timestamp mechanism. Next the implementation is described, followed by extension of the mechanism for nested DAs.

4.2.1 Additional locking rules

The new additional lock acquisition rules are summarized in Figure 4-5; they apply only if the action D that requests a lock is disconnected. The new rules prevent D from observing or affecting a state that was created by a later ordered (serial) relative. Failure to satisfy the new rules implies that D should be aborted. Note that

- Acquiring a write lock
  - D's timestamp must be no smaller than the timestamp of any lock holder on X.
- Acquiring a read lock
  - D's timestamp must be no smaller than the timestamp of any **write** lock holder on X.

Figure 4-5: Additional lock acquisition rules for a DA D on an object X

The new rules are applied only **after** the current rules approved granting a lock to a DA D; this way the new rules are applied only to conflicting locks held by ancestors of D. Note that unnecessary aborts of DAs are prevented; a DA requesting a lock on an object need not abort when a later sibling is found to hold a conflicting lock because timestamps are checked only after that sibling commits to a common ancestor or aborts. For example, the DA $A.1.D1$ (in Figure 4-1) need not be aborted if $A.2$, which has a higher (i.e., later) timestamp, is holding a lock on C_3 at the time $A.1.D1$ requested a lock. It is possible that $A.2$ would later abort, its locks would be discarded and $A.1.D1$ would be able to acquire a lock on C_3. $A.1.D1$ has to wait until $A.2$ either commits to A or aborts.

```
check_locks_for_read = proc(Reader:ainfo, OBJ:object) returns(reply) signals(abort_DA(aid))
        % Called at a site to check whether Reader can get a READ lock on OBJ
        if versions_stack$non_empty(OBJ.write_stack) then
                top_write_lock : lock := versions_stack$top_lock(OBJ.write_stack)
                if aid$non_descendant(Reader.aid,top_write_lock.aid) then
                        return(can_not_have_lock(Reader,OBJ,top_write_lock.aid))
†                     elseif aid$disconnected(Reader.aid) cand Reader.ts < top_write_lock.ts
then
†                         signal abort_DA(Reader.aid)
                     end
                end
        % Reader is a descendant of the top write locker (if any)
        if lock_set$member(Reader.aid,OBJ.read_set) then
                return(already_has_lock(Reader,OBJ))
            else
                return(can_have_lock(Reader,OBJ))
                end
        end check_locks_for_read
```

Figure 4-6: The lock acquisition rules and the additional rules applied for a (possibly disconnected) reader

Figure 4-6 and Figure 4-7 extend the algorithms presented in Figure 2-1 and Figure 2-2 to apply the new additional rules as well. The added lines are marked with a "†". Every conflicting lock is first checked by the current rules, and if it does not conflict with the request, then by the new ones. Note that only the top write lock is checked; due to the ancestry order of the write lock stack, all those below the top belong to ancestors, whose timestamps are no greater than the one of the top lock.

```
check_locks_for_write = proc(Writer:ainfo, OBJ:object) returns(reply) signals(abort_DA(aid))
        % Called at a site to check whether Writer can get a WRITE lock on OBJ
        for read_lock:lock in lock_set$elements(OBJ.read_set) do
                if aid$non_descendant(Writer.aid,read_lock.aid) then
                        return(can_not_have_lock(Writer,OBJ,read_lock.aid))
†                     elseif aid$disconnected(Writer.aid) cand Writer.ts < read_lock.ts then
†                                signal abort_DA(Writer.aid)
                        end
                end
        if versions_stack$non_empty(OBJ.write_stack) then
                top_write_lock : lock := versions_stack$top_lock(OBJ.write_stack)
                if aid$non_descendant(Writer.aid,top_write_lock.aid) then
                        return(can_not_have_lock(Writer,OBJ,top_write_lock.aid))
                     elseif Writer.aid = top_write_lock.aid then
                        return(already_has_lock(Writer,OBJ))
†                     elseif aid$disconnected(Writer.aid) cand Writer.ts < top_write_lock.ts
then
†                         signal abort_DA(Writer.aid)
                        end
                end
        % Writer is a descendant of the top write locker (if any)
        return(can_have_lock(Writer,OBJ))
        end check_locks_for_write
```

Figure 4-7: The lock acquisition rules and the additional rules applied for a (possibly disconnected) writer

4.2.2 Implementation of the Timestamp Mechanism

Timestamps are implemented as part of the local action (i.e., AINFO.ts) and as part of every lock. A possible implementation of timestamps is given in Appendix B. Messages sent for creating a handler call or committing it need also to carry the timestamp of the action (the one on the sending side).

Management of actions' timestamps

Figure 4-8 outlines the way our model is extended to maintain timestamps on actions. When any action is created, it inherits its parent's timestamp or zero if it is a topaction. The action's timestamp is incremented immediately after creating a (top) DA or a concurrent set that includes (top) DAs, hence giving an active action a timestamp higher than that of its disconnected children. When a subaction commits, if its timestamp is bigger than its parent's timestamp (i.e., it or its descendants created DAs), then the parent's timestamp must be modified to equal that of the committing subaction; thus non-blocked active actions always have a timestamp larger than the timestamp of any disconnected descendant.

- When A is created
```
if aid$topaction(A.aid) then
        A.ts := timestamp$new()
    else
        A.ts := parent(A).ts
    end
```
- When any subaction S commits to A
```
A.ts := timestamp$max(A.ts,S.ts)
```
- Immediately after A creates a DA a set of DAs and before continuing
```
A.ts := timestamp$increment(A.ts)
```

Figure 4-8: Timestamp management rules for any action A

Note how the case of concurrent siblings that created DAs is treated: The parent action (concurrent-set action), which is blocked until all its concurrent subactions terminate, gets the maximal timestamp from the committed subactions before it continues. This way the parent's timestamp will be larger than that of any DA that was created by its concurrent subactions.

Management of timestamps on locks

The rule for managing timestamps on locks is simple: Whenever an action A (whose current timestamp is $A.ts$) accesses an object O, on which A has the appropriate lock L, do:

```
L.ts := A.ts
```

Note that the rule applies in two different cases: When A first acquires the lock L, and in each of A's subsequent accesses to O[3]. The rule has to apply to actions that have already gotten

[3]This is the reason for making the distinction between "can_have_lock" and the reply "already_has_lock" in our code.

the lock because actions' timestamps can increase during their lifetime. The cost of updating timestamps on every access is negligible since the appropriate lock is checked on every access anyhow (see the requirements on page 24).

Separation of read and write accesses, and the requirement to have the appropriate lock for each[4], prevent some unnecessary aborts of DAs. For example, suppose some action A with timestamp 1 writes to an object O, and later reads O when its timestamp is 3. A's disconnected child $A.D5$ with a timestamp 2 should be able to read O at this point, but not modify it. Unless A has separate locks, one for each operation, $A.D5$ may be denied a read-lock on O if the timestamp on A's write lock reflects A's read access (with timestamp 3).

```
inherit_read_lock = proc (Ancestor, Locker: aid, OBJ:object)
    % Requires: Locker descendant of Ancestor, Locker has a read-lock on OBJ
    % Modies: OBJ.read_set and Locker's read lock there
    % Eects: Ancestor inherits Locker's read-lock on OBJ
    locker_lock : lock := lock_set$nd(Locker,OBJ.read_set)
    locker_lock.aid := Ancestor
    ancestor_lock : lock := lock_set$nd(Ancestor,OBJ.read_set)
        except when not_found: return end
    locker_lock.ts := timestamp$max(ancestor_lock.ts,locker_lock.ts)
    lock_set$remove(Ancestor,OBJ.read_set)
    end inherit_read_lock
```

Figure 4-9: Ancestor inherits Locker's read-lock on OBJ

When the locks of an action are inherited by an ancestor, the timestamps on the locks are not affected by the ancestor's current Timestamp (i.e., AINFO.ts). The timestamp is a part of the lock (like the version), and only the lock's owner is replaced. Figure 4-9 describes how a read-lock is inherited by an ancestor. Basically the lock owner is replaced, and if the ancestor already has a read-lock, it is discarded. The maximal timestamp of the two locks prevails; this way the latest read access on behalf of the ancestor is reflected.

Figure 4-10 describes how a write-lock is inherited by an ancestor. The inherited lock, with its version and timestamp, replaces all the locks below in the stack up to (and including) the ancestor's lock. Note that the timestamp on the write-lock reflects the time its version was last modified, regardless of the ancestor's (possibly higher) timestamp at the time of the inheritance.

[4] I.e., a read-lock is needed before reading even if the action already has a write-lock there, see page 24.

```
inherit_write_lock = proc (Ancestor, Locker: aid, OBJ:object)
        % Requires: Locker descendant of Ancestor.
        %                  Locker has the top write-lock in OBJ.write_stack
        % Modies: OBJ.write_stack
        % Eects: Ancestor inherits Locker's write-lock (with version) on OBJ
        locker_lock : lock := versions_stack$top(OBJ.write_stack)
        top_aid : aid := locker_lock.aid
        while aid$descendant(top_aid,Ancestor) do
                versions_stack$pop(OBJ.write_stack)
                top_aid := versions_stack$top(OBJ.write_stack).aid
                end
        locker_lock.aid := Ancestor
        versions_stack$push(OBJ.write_stack,locker_lock)
        end inherit_write_lock
```

Figure 4-10: Ancestor inherits Locker's write-lock on OBJ

4.2.3 Timestamps for Nested DAs

Our timestamp mechanism can be extended to handle the case of more than one level of disconnection (i.e. nested DAs). Such a case is shown in Figure 4-11; the disconnected action $A.2.D2$ created another DA, $A.2.D2.D1$. The difference between this case and the case of a single disconnection level is that a nested DA like $A.2.D2.D1$ has to be synchronized with an ancestor that is itself disconnected. A timestamp represented as a single numerator does not suffice here, and has to be extended to a multi-part timestamp. The new lock acquisition rules, on the other hand, are basically the same.

The extended timestamp mechanism employs a separate timestamp for every level of disconnection. A disconnected action D at some level k of disconnection (i.e., there are k disconnections on the tree path between D and the root topaction) keeps a list of $k + 1$ timestamps. For example, when the DA $A.2.D2$ (in Figure 4-11) is created, it gets its parent's timestamp of 1 with a new level (initially 0) added, represented as 1.0 . Two (Multi-level) timestamps are compared by matching them level by level, left to right; the first level with non-equal values determines the order (a short timestamp is padded with zeroes on its right side). (Disconnected) actions update only the right-most part of their timestamp when they create DAs. For example, $A.2.D2$'s timestamp is changed from 1.0 to 1.1 after creating a DA. The additional part in

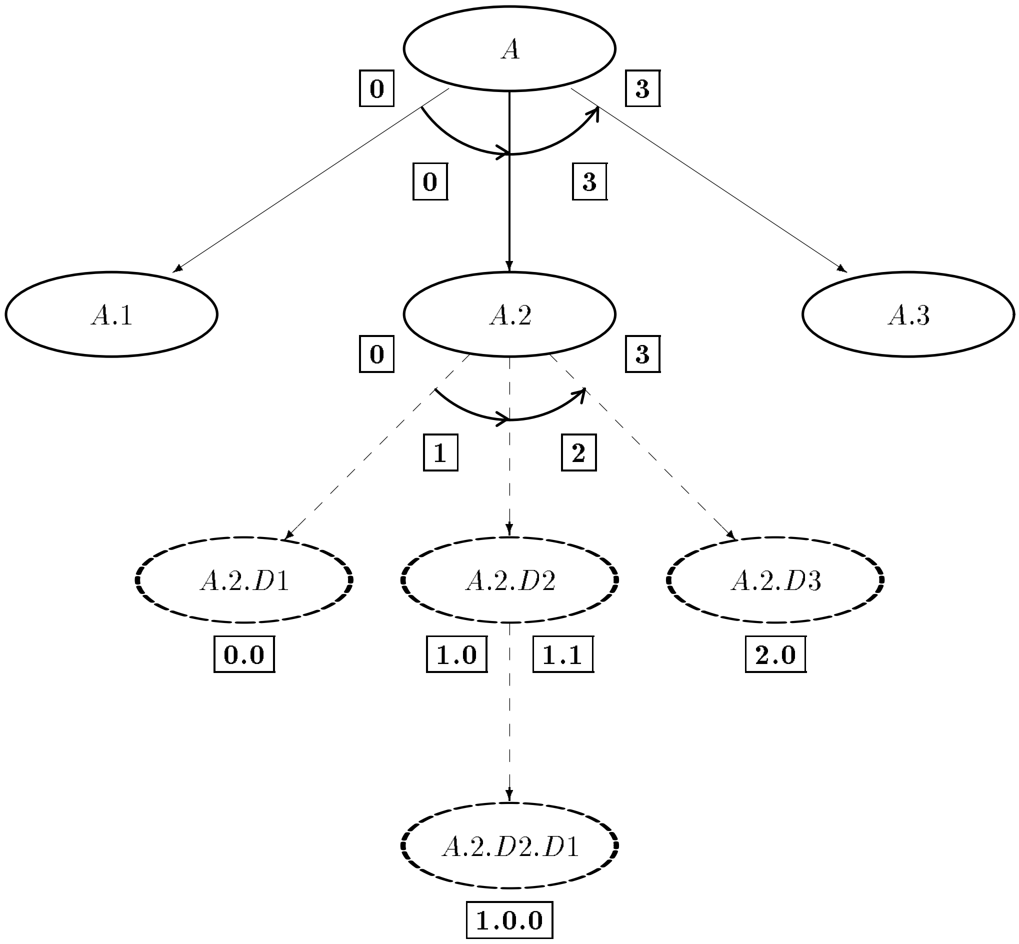


Figure 4-11: An example for serializing nested DAs using timestamps

the timestamp has no effect on the way committing children update their parents' timestamps because the difference in the number of parts occurs only between parents and top DA children, who do not commit normally anyhow.

The timestamp rules and implementation presented in this chapter apply to nested DAs as well. The multi-part details are encapsulated inside the abstraction for timestamps (see the implementation in Appendix B). Timestamp operations like **increment**, **max**, **equal**, etc., handle the appropriate parts of the representation internally as needed.

4.3 Concurrent DAs

Disconnected actions that were created by concurrent subactions (concurrent DAs) pose problems that can not be solved by a mechanism like timestamps. The timestamp mechanism proposed above enforces the static creation order of the serial subactions upon their disconnected actions. Concurrent subactions, on the other hand, are serialized dynamically. They are created simultaneously, run concurrently and should behave as though they were run in some serial order. This order reflects the order in which the subactions accessed shared atomic data and is determined dynamically as the actions run.

Execution of an action that uses concurrent DAs may result in an inconsistent state. The scenario described in Section 4.1 can take place when the two subactions, $A.1$ and $A.2$, are created simultaneously as concurrent siblings (and have initially the same timestamp). The locking rules presented above do not prevent the non-serializable execution; $A.1$ updates C_1 and C_2, commits and A inherits its locks, and $A.2$ acquires the locks held by A, does its work and commits to A. The DA $A.1.D1$ can acquire the lock on C_3 because A inherited it from $A.2$. Timestamps are not sufficient in this case, and no information exists at C_3's site (assume that the other actions run at other sites) to tell whether $A.1.D1$ can get a lock on C_3 or not. By acquiring the lock on C_3, $A.1.D1$ serializes its concurrent parent $A.1$ after its sibling $A.2$ that affected C_3 before, resulting in a non-serializable execution of $A.1$ and $A.2$ since C_2 was accessed in the opposite order. Note that the serialization order is determined by the order in which locks were passed from one relative ($A.1$) to another ($A.2$) via the LCA (A); our solution takes advantage of this fact to monitor the serialization of concurrent subactions serially at the site of the LCA.

In a model without DAs, the commit order of concurrent subactions is a possible serial order of the subactions; the exact serialization order is usually only a partial order consistent with the commit order. When DAs are used, the actual commit order of the concurrent subactions may not be consistent with the way they get serialized; a disconnected descendant of a concurrent subaction S may be active **after** S's commit and force serialization constraints on S by acquiring locks on objects that were accessed by other concurrent siblings of S.

This chapter proposes a new mechanism that uses *Constraint Tables* and *Fingerprint Locks* to control access of disconnected actions to objects locked in conflicting mode by a concurrent relative. The new mechanism monitors the serialization order of the concurrent subaction, and keeps an explicit record of it (as a Constraint Table); this record is consulted as needed whenever a DA tries to acquire a lock.

The idea behind the new mechanism is to have centralized control over the serialization of concurrent subactions. Our problem is analogous to the deadlock detection problem in distributed systems. A deadlock implies a cycle of blocked concurrent activities, each waiting for the next one; a non-serializable execution of concurrent atomic actions can happen if, by creating serialization constraints (by accessing shared objects and using DAs), a cycle of actions is created where each action is serialized after the next. Our solution is similar in principle to the deadlock detection algorithm that uses a centralized *wait-for graph* ([Gray 1978]) to monitor wait-for relations and prevent a cycle.

The new mechanism does have a significant performance cost, but it is a very permissive mechanism that ensures serial behavior. Other mechanisms can be devised that have a lower cost, but they may abort many DAs unnecessarily. For example, another mechanism can be designed to totally prevent DAs from acquiring conflicting locks on objects that were used by concurrent relatives; such a mechanism need not track serialization order, but is too restrictive.

The rest of this section is organized as follows: We begin the description of the new mechanism by presenting a simplified version of our solution, and continue by describing how we make the simplified solution practical. Next we give details of the way our solution can be implemented, and end with discussion of several issues that were ignored in our solution.

4.3.1 A simplified solution

The simplified solution, presented below, to the problem of serializing concurrent subactions that create DAs, ignores costs (i.e., extra messages, time and space) and failures (i.e., crashes and aborts); it focuses on the basic method to monitor serialization of concurrent subactions and control access of concurrent DAs to objects.

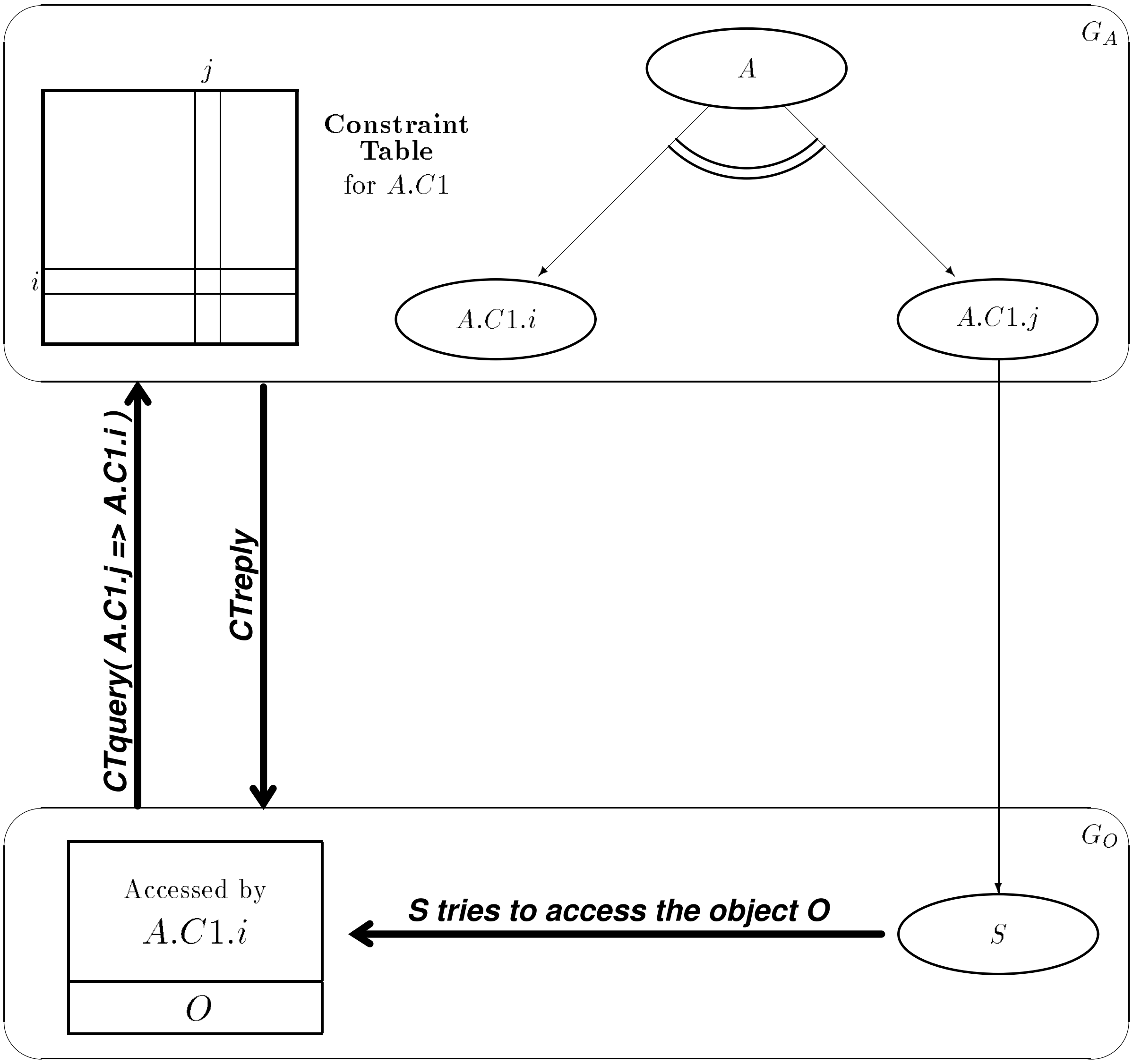
Sites maintain a *Constraint Table (CT)* for every concurrent-set action created locally. The CT can be represented by an $n \times n$ matrix, where n is the total number of concurrent subactions in the set, and an entry $CT(i, j)$ describes the serialization order between subactions i and j ("before", "after" or has some initial value). The CT represents a directed graph in which each vertex is a concurrent subaction, and each edge is a serialization constraint. A cycle in the constraint graph is equivalent to a non-serializable execution of the concurrent subactions ([Eswaran, *et al.* 1976]).

We assume that every atomic object remembers its history. For example, an object O that was modified by a subaction S_1 remembers S_1, enabling O's site to know that a subsequent access of a subaction S_2 to O (in conflicting mode) implies the serialization constraint "S_2 ordered after S_1".

Whenever any subaction S_2 tries to acquire a lock on an object that was previously accessed by a (committed) concurrent relative S_1 (in conflicting mode), a **special** query message, *CTquery*, has to be sent from the object's site (G_O) to G_{LCA}, the site of (the concurrent-set action) $LCA(S_1, S_2)$, specifying the possible constraint "S_2 ordered after S_1". When G_{LCA} receives the *CTquery*, it adds the serialization constraint to the CT of $LCA(S_1, S_2)$ and acknowledges with a *CTreply* message. Hence the CT maintains explicitly the partial serialization order of its concurrent subactions.

A *CTquery* on behalf of a regular subaction S_2 is only used for monitoring serialization; for a DA D the *CTquery* message and its acknowledgment are also used to control the DA's access to objects. When an access by D is found to create a cycle in the constraint graph, the CT's site does not update the CT and uses the *CTreply* message to inform the D's site that access should be denied (i.e., "abort D"), otherwise the CT is updated and the *CTreply* approves of D's access.

Figure 4-12 illustrates the simplified solution. When a subaction S tries to access an object



G_A
j
i
Constraint Table for $A.C1$
A
$A.C1.i$
$A.C1.j$
CTquery(A.C1.j => A.C1.i)
CTreply
G_O
Accessed by $A.C1.i$
O
S tries to access the object O
S

Figure 4-12:

O at site G_O, O if found to have been used before by a concurrent relative of S. A *CTquery* message is sent from G_O to the site G_A of $LCA(S, A.C1.i)$, describing the new constraint "$A.C1.j$ must be serialized after $A.C1.i$". The constraint table at G_A is updated and a *CTreply* is returned. If S had been a disconnected action, then the CT would be checked first, and the *CTreply* would return the decision.

4.3.2 Making the solution practical

The simplified solution proposed above works, but several efficiency issues must be dealt with to make this solution practical. In the following we discuss the major issues: How to reduce the number of messages, how to avoid the overhead when DAs are not used and how to ensure that objects would have their relevant access history available when locks are acquired.

Reducing the number of messages

Our simplified mechanism requires exchange of messages for every access of an action to an object that was used by a concurrent relative. We propose here to reduce the number of messages by caching previous replies locally and using other existing messages to carry some of the data needed by our mechanism.

The use of *CTquery* messages is parallel in many cases to the use of regular query messages (to the LCA). Basically all that is needed to make a regular query message into a *CTquery* message is the name (AID) of the action that tries to access the object. Our solution unifies the two kinds into one. Though this conversion of regular queries (and replies) seems to come with no cost, we do lose a possible optimization: Regular queries are not sent on behalf of specific actions accessing specific objects; therefore one query may be sent on behalf of several actions trying to use several objects. We can still use this optimization when a *CTquery* is not needed.

Some *CTqueries* can not "catch a ride" on existing queries. In some cases a regular query is not needed, but giving a lock still implies the creation of serialization constraint (see *Fingerprint Locks* below). We send more (modified regular) queries in these cases, but may refer to them as *CTqueries* to stress their purpose.

Local caching can help avoid some messages. The idea is: The site replying with *CTreply* will add the relevant CT to the reply, and the site receiving the reply will cache that CT locally, possibly replacing an older copy. This way, if the local copy is consulted first, the message exchange can be avoided when the order is known locally.

Enable the mechanism only when DAs are used

As stated before, one of our design goals is to have the new model perform similarly to the current one when DAs are not in use. Having a constraint table for every concurrent set of subactions and sending CT messages does not contribute to this goal. We would like the CT mechanism to be *triggered* only when it is needed, i.e., after some concurrent subaction created DAs. The problem is that the site of the concurrent-set action C has no a priori knowledge about C's intention to create DAs; we need a way to detect it dynamically.

We choose the triggering event to be the commit of the first concurrent subaction S with a timestamp bigger than its ancestor's, the concurrent-set action C. This event implies at least one concurrent subaction (S) created DAs, and our mechanism is needed (except when S is the last concurrent subaction to commit to C). Triggering the CT mechanism means not only creating a new CT, but also notifying sites that make relevant queries that CT caching has to be done and that lock propagation must be handled differently (see below).

When S commits and triggers the mechanism for C, a new constraint table is allocated at C's site for C. This CT needs not handle those concurrent subactions of C that already committed[5], with the exception of S, which also gets an entry in the CT.

Once the mechanism has been triggered (for C), the "committed to LCA" reply to every query regarding C's subactions (replied from C's site) must be tagged with the tag *make_FP* and carry a recent copy of the CT. A site receiving a tagged reply would cache the CT copy locally and handle lock propagation differently (as described below).

Note that C's site may notice that DAs were created on C's behalf before S commits by receiving a query from (or about) some DA, descendant of C. The mechanism need not be triggered at this point because non-serializable execution of concurrent subactions using DAs must include a commit of some subaction using DAs[6].

[5] Because they created no DAs.

[6] The execution is serializable prior to the commit of the first concurrent subaction that created DAs because no concurrent subaction has yet violated the rule of strict two-phase locking; i.e., acquired a lock (by a descendant DA) after releasing another.

Identify the last action(s) to access an object

The simplified solution required that the history of each object be kept, so the last (concurrent) action to access the object in conflicting mode can be known. Our model does keep this information with each object in the form of locks, but once locks are inherited, the identity of the real accessor is no longer known. More specifically, after a lock that was held by a descendant of a concurrent subaction S is inherited by (an ancestor of) the concurrent-set action C, it is impossible to tell which of C's concurrent subactions had it. We need a way to remember the last (concurrent) subaction that modified some object, and possibly several that read the object (and are serialized after that writer).

Our solution is to both keep that inherited lock and inherit it. This can be done as follows: Inherit locks as before, but once a lock is propagated to the level of the concurrent-set action, a copy of the old lock is kept as a *Fingerprint Lock (FPL)*. This way, when a subaction tries to get a lock on an object previously used by some concurrent relative, it would be known with whom it conflicts. The FPL is a regular (read or write) lock, but is tagged with the boolean flag *fp*. The FPL is transparent to operations that do not need it (e.g., lock query by a later serial relative or commit of the transaction[7]), and is used only by our mechanism.

Our solution actually requires a FPL to exist on an object before granting a conflicting lock; that is, first queries are sent, and their replies cause such conflicting locks to propagate and also become FPLs, next checks are done to ensure the correct serialization. This leads to a simple (and uniform) additional lock granting rule (and implementation): "The local CT must approve the ordering constraint with every conflicting FPL before the lock is granted".

We could generate FPLs whenever a lock is inherited (as explained above) and keep it until the transaction commits, but this may take a performance toll. Our implementation begins generating FPLs only after the CT mechanism has been triggered (for the appropriate concurrent set of subactions); that is, FPLs would be created only if the reply is tagged with *make_FP*. FPLs are also discarded once they are no longer needed; we describe below how obsolete write FPLs are discarded, as a part of maintaining the write stack, and ignore for now the need to discard read FPLs, as obsolete read FPLs only slow our algorithm, but do not

[7]We ignore FPLs in the description of the commit mechanism (Chapter 5) as they have no effect there.

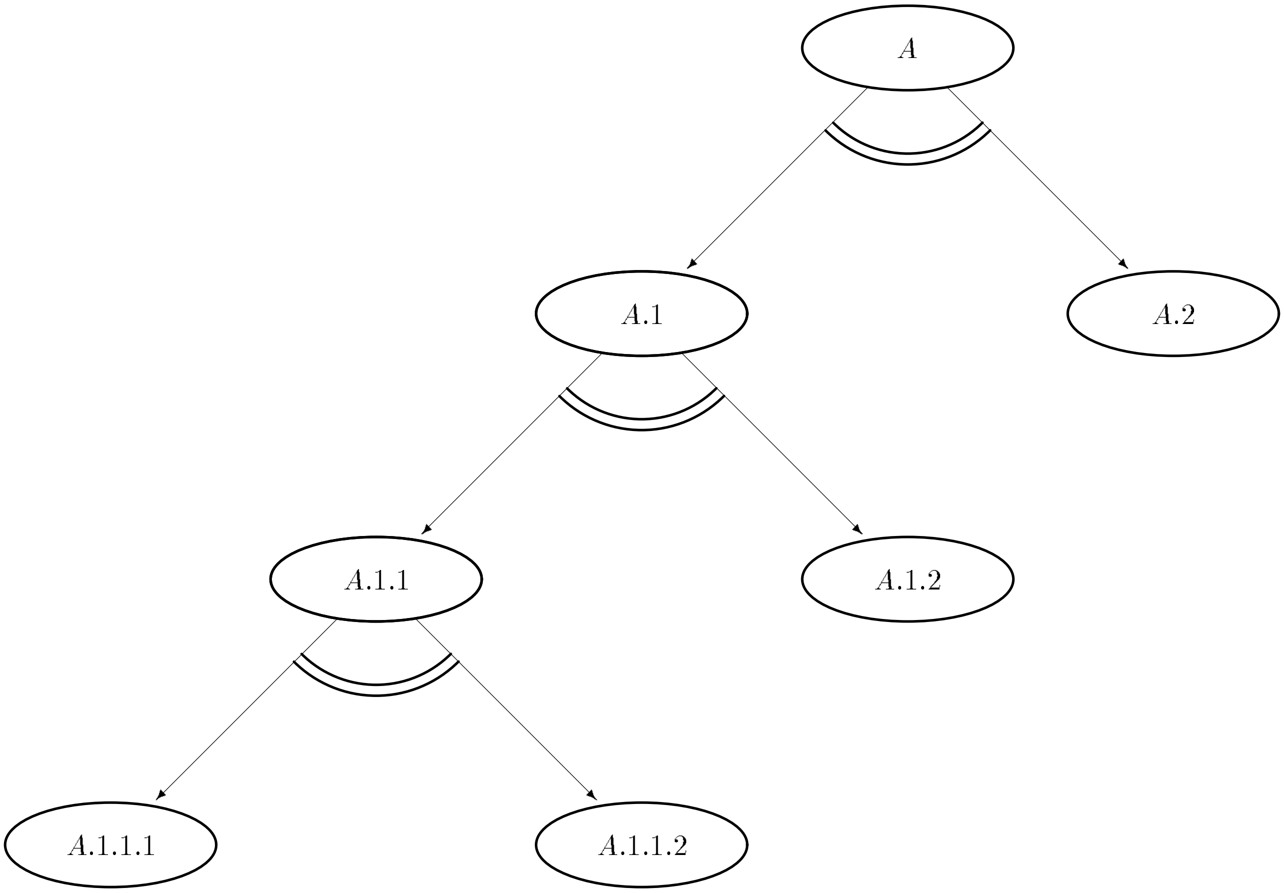


Figure 4-13: An example of multiple levels of concurrent subactions.

change its behavior.

Note that propagation of a lock from a holder H to an ancestor A can generate more than one FPL; when there are several concurrent-set actions between H and A (including A), the "arm" (concurrent subaction) in each must be remembered. This applies to both read and write locks.

Such a case is illustrated in Figure 4-13. The subaction $A.1.1.2$ created DAs and caused $A.1.1$'s CT to be triggered. Cascaded triggering occurs when $A.1.1$ commits to $A.1$ and later to A. After that, if the subaction $A.2$ tries to get a (conflicting) lock on some object O that is locked by $A.1.1.1$, the query sent to A's site indicates that with "$A.1$ committed up to A", and tagged with *make_FP*.

Creating a FPL on O for $A.1$ alone is not enough, because $A.2$ may abort after that and a DA D created by $A.1.1.2$ may try later to get a lock, but no trace of $A.1.1.1$'s access would then be found on O. The solution is to create a FPL for every concurrent subaction between the lock holder and the LCA. In our example, when the tagged reply arrives, three FPLs are needed: for $A.1$, $A.1.1$ and $A.1.1.1$.

Once a read FP lock is created, it is put in the unordered set of read-locks. Multiple read FPLs (of the same concurrent level) may exist on an object because readers do not enforce serialization constraints among themselves. For simplicity, our implementation assumes that read FPLs are not discarded unless some ancestor aborted or their transaction committed.

When a write FP lock is created, it is put on top of the write stack; when several FPLs are created, they are left on the stack in ancestry order. See Figure 4-14 for the state of the stack in the case of the previous example. When an action like $A.2$ acquires a write lock, it will be positioned above the FPLs in the stack, so if $A.2$ aborts, its lock is discarded and other actions (e.g., concurrent siblings of $A.1$ and $A.2$) can still use the FPLs. If $A.2$ commits and A inherits its write lock, then the three FPLs (shown in Figure 4-14) become obsolete, and are discarded, but a new FPL is created for $A.2$. Note that our implementation of "inherit_write_lock" (page 53) already discards all these locks since it discards all write-locks (of any kind) between the inherited lock (e.g., $A.2$'s lock on the top) to the position of the ancestor (A).

In general, the sequence of real locks on the write stack will be interleaved with some subsequences of FPLs. The real locks represent the current path P in the action tree, while the subsequences of FPLs represent previous paths branching away from P that may become useful if actions with locks above them abort. When the top lock is a real lock, regular work is done with it, ignoring possible FPLs below. But when several actions abort, the FPLs below may get exposed.

4.3.3 Implementation

Below we describe the fine details needed to implement our mechanism. We assume that sites keep CTs and manage them, updating copies as tagged replies arrive and discarding them when the transaction terminates[8]. Queries are assumed to contain the AID of the action A requesting

[8]CTs can be kept in a data structure similar to ACTIVE or COMMITTED.

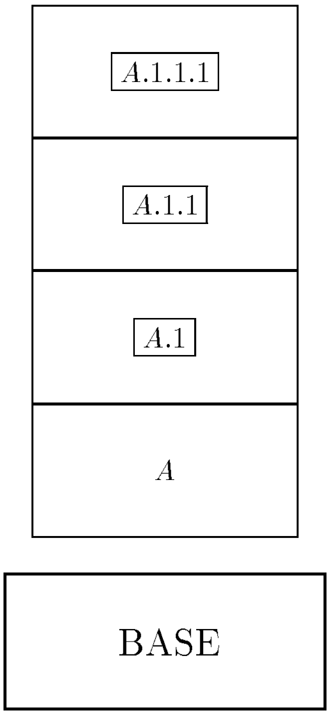


Figure 4-14: The state of the write stack with FP locks (boxed).

the lock, and are sent whenever A can not get a lock due to a conflicting one.

When a concurrent subaction S commits to its concurrent-set action C (which must happen locally), if S.ts is bigger than C.ts then a new CT is created for C locally (if not yet so done) to map all C's subactions that are still active, including S. The implementation must be able to tell at any time which of C's subactions have not yet terminated.

When a lock query arrives at C site, G_C, specifying that a descendant of a concurrent subaction S_2 requests a lock held by S_1 (such that $C = LCA(S_1, S_2)$), then if S_1 has committed to C, and C has a CT, then that CT is updated if necessary, unless the update creates a cycle. Then G_C replies, tagging its "committed to C" reply with *make_FP* and adding a copy of the CT to it.

When a tagged reply arrives, implying that the locks of a locker S are to be propagated to an ancestor A, the locks are propagated to A, but FPLs are created for every concurrent subaction on the path between S and A (S included, A excluded). Figure 4-15 describes the routine to be called when a tagged reply arrived (for a non-tagged reply, the one in Figure 4-9 is used), and Figure 4-16 gives the routine for write locks (the one in Figure 4-10 is used when non-tagged replies arrive).

```
make_FP_read = proc (Ancestor, Locker: aid, OBJ:object)
        % Called when the reply is tagged, instead of "inherit_read_lock"
        % Requires: Locker descendant of Ancestor, Locker has a read-lock on OBJ
        % Modies: OBJ.read_set
        % Eects: Ancestor inherits Locker's read-lock on OBJ, and FPLs are
        %             created for all the concurrent subactions between them.
        locker_lock : lock := lock$copy(            % inherit_read_lock modies the original
                lock_set$nd(Locker,OBJ.read_set))           % return a reference to that lock
        inherit_read_lock(Ancestor,Locker,OBJ)
        for conc_sub:aid in aid$concurrent_subactions_between(Ancestor,Locker) do
                FPL : lock := lock$copy(locker_lock)
                FPL.aid := conc_sub
                FPL.fp := true
                lock_set$insert(FPL,OBJ.read_set)
                end
        end make_FP_read
```

Figure 4-15: Ancestor inherits read-locks and creates read FPLs.

```
make_FP_write = proc (Ancestor, Locker: aid, OBJ:object)
        % Requires: Locker descendant of Ancestor.
        %                   Locker has the top write-lock in OBJ.write_stack
        % Modies: OBJ.write_stack
        % Eects: Ancestor inherits Locker's write-lock (with version) on OBJ,
        %             and FPLs are created for all concurrent subactions between them.
        locker_lock : lock := versions_stack$top(OBJ.write_stack)
        inherit_write_lock(Ancestor,Locker,OBJ)
        for conc_sub:aid in aid$concurrent_subactions_between(Ancestor,Locker) do
                FPL : lock := lock$copy(locker_lock)
                FPL.aid := conc_sub
                FPL.fp := true
                versions_stack$push(OBJ.write_stack,FPL)
                end
        end make_FP_write
```

Figure 4-16: Ancestor inherits write-locks and creates write FPLs.

The algorithm for deciding on lock grant becomes simple due to the fingerprint scheme shown above; when an action tries to acquire a lock on an object, conflicting locks of concurrent relatives are replaced (if our mechanism was triggered) with FP locks, and the serialization constraint can be known locally. If by that time relevant information has already arrived at the local CT, the lock granting process can continue, else an extra query is needed.

```
serialized_after = proc (S,locker:aid) returns(bool) signals(abort_DA(aid))
        % Called when subaction S accesses an object with an FP lock of locker.
        % Makes a decision based on the relevant CT.
        % Eects: Returns true if S is serialized after locker in the relevant
        %          local CT or if the two are not concurrent relatives.
        %                    MEANING: S can ignore this FPL.
        %          Returns false if no order exist in the CT or CT not found.
        %                    MEANING: A CTquery need to be sent to nd the order.
        %          Signals abort_da if S is serialized before locker in the CT.
        %                    MEANING: S must be a DA out of serial order.
        if aid$not_concurrent_relatives(S,locker) then return(true) end
        relevant_ct : ct := nd_relevant_ct(aid$LCA(S,locker))
           except when not_found: return(false) end
        if ct$after(S,locker) then return(true)
              else signal abort_DA(S)
              end except when no_order: return(false) end
           end
end serialized_after
```

Figure 4-17: Check local Constraint Table when a lock check finds an FP lock

Figure 4-17 presents an abstraction that decides for an action A finding a conflicting FP lock held by B, based on local CT information, which of the three cases applies: *"Ignore this FPL"* when the ordering constraint conforms to the CT data (or when A and B are not concurrently related), *"CTquery is required"* when ordering information does not exist locally, and *"abort this DA"* when A is (a DA) out of order.

The routine serialized_after is used by our lock checking procedures, presented in Figures 4-18 and 4-19. These procedures do the complete checking process, combining the current lock acquisition rules, the additional rules that use timestamps, and the rules that use FPs. Note that these are the procedures presented before (in Figures 4-6 and 4-7) with additional lines (marked with "‡") that do the work of checking FP locks.

Whenever one of the two procedures replies with "can_not_have_lock", a query is sent to the site of the LCA, regarding the conflict between the action A that tries to access the object and the conflicting lock owner. The reply to the query, if tagged with *make_FP*, calls the lock

```
check_locks_for_read = proc(Reader:ainfo, OBJ:object) returns(reply) signals(abort_DA(aid))
        % Called at a site to check whether Reader can get a READ lock on OBJ
        if versions_stack$non_empty(OBJ.write_stack) then
                top_write_lock : lock := versions_stack$top_lock(OBJ.write_stack)
‡               while top_write_lock.fp do % scan FPLs top down
‡                       if serialized_after(Reader.aid,top_write_lock.aid) then
‡                               top_write_lock := top_write_lock.next_below
‡                               else return(can_not_have_lock(Reader,OBJ,top_write_lock.aid))
‡                               end resignal abort_DA
‡                       end
                if aid$non_descendant(Reader.aid,top_write_lock.aid) then
                        return(can_not_have_lock(Reader,OBJ,top_write_lock.aid))
†                       elseif aid$disconnected(Reader.aid) cand Reader.ts < top_write_lock.ts
then
†                           signal abort_DA(Reader.aid)
                        end
                end
        % Reader is a descendant of the top write locker (if any)
        if lock_set$member(Reader.aid,OBJ.read_set) then
                return(already_has_lock(Reader,OBJ))
            else
                return(can_have_lock(Reader,OBJ))
                end
        end check_locks_for_read
```

Figure 4-18: The complete modified lock acquisition rules applied when a (possibly disconnected) action requests a **read** lock on the object OBJ.

```
check_locks_for_write = proc(Writer:ainfo, OBJ:object) returns(reply) signals(abort_DA(aid))
        % Called at a site to check whether Writer can get a WRITE lock on OBJ
        for read_lock:lock in lock_set$elements(OBJ.read_set) do
                if aid$non_descendant(Writer.aid,read_lock.aid) then
‡                       if read_lock.fp then
‡                               if serialized_after(Writer.aid,read_lock.aid) then continue
‡                                       else return(can_not_have_lock(Writer,OBJ,read_lock.aid))
‡                                       end resignal abort_DA
‡                           else % this lock is not FPL
                                return(can_not_have_lock(Writer,OBJ,read_lock.aid))
‡                           end
†                       elseif aid$disconnected(Writer.aid) cand Writer.ts < read_lock.ts then
†                                   signal abort_DA(Writer.aid)
                          end
                end
        if versions_stack$non_empty(OBJ.write_stack) then
                top_write_lock : lock := versions_stack$top_lock(OBJ.write_stack)
‡               while top_write_lock.fp do % scan FPLs top down
‡                       if serialized_after(Writer.aid,top_write_lock.aid) then
‡                               top_write_lock := top_write_lock.next_below
‡                               else return(can_not_have_lock(Writer,OBJ,top_write_lock.aid))
‡                               end resignal abort_DA
‡                       end
                if aid$non_descendant(Writer.aid,top_write_lock.aid) then
                          return(can_not_have_lock(Writer,OBJ,top_write_lock.aid))
                      elseif Writer.aid = top_write_lock.aid then
                          return(already_has_lock(Writer,OBJ))
†                       elseif aid$disconnected(Writer.aid) cand Writer.ts < top_write_lock.ts
then
†                           signal abort_DA(Writer.aid)
                          end
                end
        % Writer is a descendant of the top write locker (if any)
        return(can_have_lock(Writer,OBJ))
        end check_locks_for_write
```

Figure 4-19: The complete modified lock acquisition rules applied when a (possibly disconnected) action requests a **write** lock on the object OBJ.

propagation routines described above. When A later tries again[9], another query may need to be generated.

4.3.4 Discussion

Below we discuss some of the remaining issues, most of which were ignored previously to maintain comprehensibility.

Discarding obsolete FP read-locks

As described above, once a FP read lock is added to an object, it remains there until its transaction terminates (or some ancestor aborts). Some FP read-locks may become obsolete later in the execution; lock requests would not be denied due to obsolete FP read-locks, but they would need to be checked on every write access and also consume space.

A FP read-lock L becomes obsolete if any lock holder H (read or write), serialized after L, commits to an ancestor of L. The reason is simple: Later write accesses would have to be serialized after H, thus after L's holder anyway. In summary, a FP read-lock L held by a subaction S on an object O, can be discarded when

- Any write-lock on O is inherited by an ancestor of S.
- Another read-lock on O, held by a later serial relative of S is inherited by an ancestor of S.
- S's transaction commits.
- Some ancestor of S aborts.

Not delaying a regular subaction that created a constraint

In our mechanism, when a regular subaction S finds a conflicting FP lock held by a concurrent relative R, and the ordering constraint "S after R" is not found in the local CT, a query (actually *CTquery*) has to be sent to the site of $LCA(S, R)$ to register that constraint.

[9] The implementation can use either busy-waiting or put A in a waiting queue and reactivate it after locks propagate.

S need not be delayed until the *CTreply* comes; S is a regular action and the current locking rules already ensure its serialization. However, the following rare scenario may take place if S is not delayed: S requests a lock that conflicts with a FP lock held by the concurrent relative R and a query is sent, then S finishes quickly and commits to the concurrent-set action, before the query message had gotten there. Next a disconnected descendant of R may request one of S's locks and get it because the ordering constraint has not yet reached the CT.

A solution to the problem that prevents the delay of S is to piggyback the constraints with the regular action that created them, and check the CT of the concurrent-set action C when S commits up to C.

Handling failures

Our mechanism need not care about site crashes; when a site that was visited by descendants of a concurrent-set action C that committed to C crashes (before C's transaction prepares), C becomes an orphan. Subactions that abort, however, do pose a problem.

When some subaction A, a proper descendant of a concurrent subaction S of C, creates a serialization constraint in C's CT (by requesting a lock someplace) and aborts, its constraint may become invalid. Such erroneous constraints can cause unnecessary aborts of DAs that try to violate the erroneous order. Note, however, that an abort of a concurrent subaction like S causes no harm; while S was active, no other sibling could have been serialized after S.

The solution presented above ignores the possibility of an abort of a subaction A that created a constraint in the CT. (The price to be paid is potential unnecessary aborts of DAs when subactions like A abort). In the following we outline a way to handle aborts; it was not implemented to avoid additional complexities in the presentation of the mechanism.

The idea behind handling aborts is keeping an accurate record of every serialization conflict created, and discarding this record when the action that created that conflict (or one of its ancestors) abort.

Our simplified mechanism (without caching of CTs) can implement the above idea by implementing the CT entries as counters. Every constraint created is reported to the central CT by a *CTquery* message; if the *CTreply* approves the order, the appropriate counter would be incremented. The new lock that is acquired after the reply is received is marked to refer to that CT. When a subaction A is aborted, and its locks are released, then for any marked lock A had, a special message would be sent to the appropriate CT and the appropriate counter would be decremented. This way, when a counter reaches its initial value again, the serialization conflict would be gone. More work is needed, however, to adapt this scheme to our optimized mechanism (which uses caching, etc.).

4.4 Conclusion

The lock acquisition and propagation scheme presented above ensures serial behavior of nested atomic actions systems that use disconnected actions. The above algorithms were simulated successfully, but no performance measurements were taken. The performance is expected to depend heavily on the various optimizations, some of which were mentioned above, which were not implemented.

The above scheme acquires the requested lock by checking all the conflicts with existing locks and applying the serial or concurrent rules, depends on the ordering relation in each conflict. Our choice of using two mechanisms was derived by considering costs; the timestamps mechanism is almost free, in comparison to the CT/FP scheme that carries a considerable price. Using the CT mechanism alone to handle both cases is possible but too expensive.

The case of nested DAs was elaborated for the serial DAs. It should make no difference for concurrent DAs because the point is to find the serialization constraint between the concurrent subactions, regardless of the level of disconnection.

Note also that the case of concurrent subactions, all or some disconnected, (e.g., in Figure 6-1) is handled correctly by using timestamps alone; DAs running concurrently still obey the strict two-phase locking protocol. However, if any of the concurrent siblings (DAs) creates (nested) DAs, the CT mechanism has to be used.

Chapter 5

Commit with Disconnected Actions.

This chapter discusses how a nested atomic action system using disconnected actions would commit its transactions. Disconnected actions, which were introduced into our model to exploit concurrency and improve performance, violate some of the preconditions required by the commit mechanism: Some (disconnected) actions may still be active, and information needed by the coordinator of the commit protocol may be scattered around the system.

This chapter describes a modified mechanism that ensures that a system using DAs would have the desired commit semantics. The new commit mechanism aborts active (disconnected) actions, collects the missing information and makes the effects of the non-aborted topaction and all its non-aborted descendants permanent.

We shall start by discussing how to modify the existing Two Phase Commit protocol to handle the various situations that may occur when disconnected actions are used, and examine (in Section 5.2) in detail how atomic objects at a site that participated in the transaction are to be handled during the execution of the commit protocol.

5.1 The modified Two Phase Commit Protocol

The design of the Two Phase Commit protocol, presented in Chapter 2, assumes that the site where the topaction runs received all the necessary information by the time the topaction decided to commit. This information includes the names of all the sites that participated in that transaction, with whom the topaction's site (the coordinator) is to perform the protocol, and

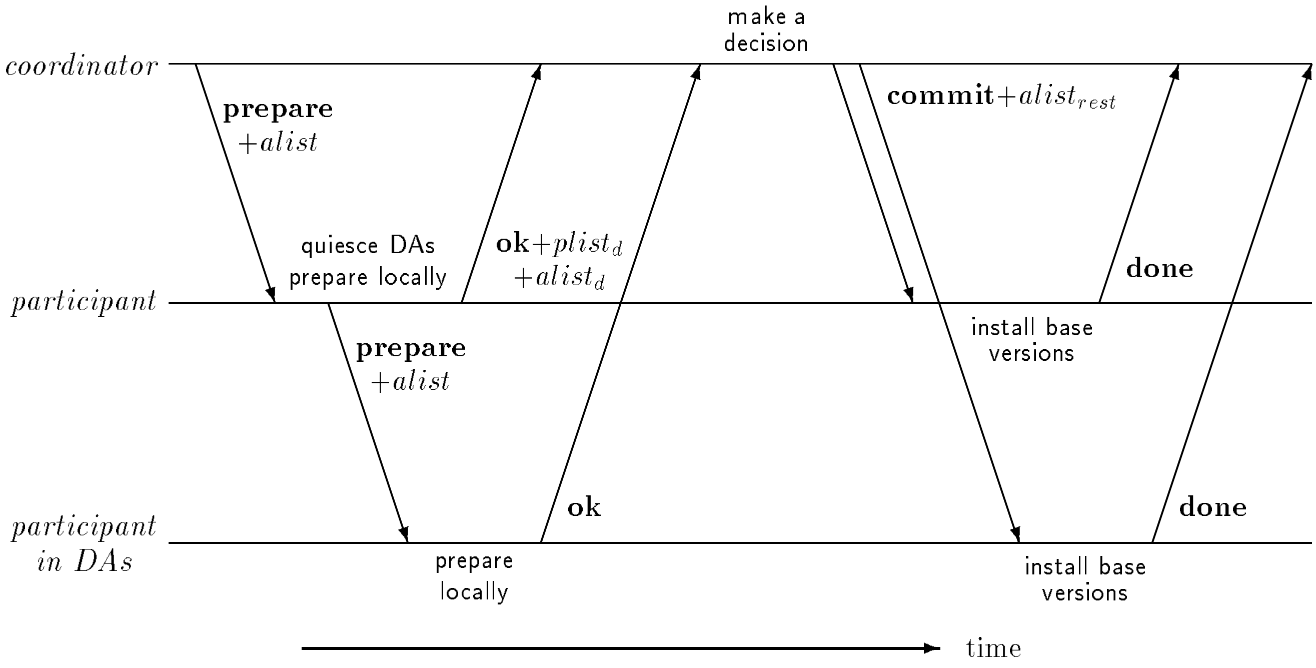


Figure 5-1: A time-space diagram of the modified Two Phase Commit protocol.

the names of all the subactions that aborted, required by the participating sites for determining the correct states of their objects.

For a nested atomic action system with disconnected actions the previous assumption may not hold. When a topaction decides to commit it may still have active disconnected descendants that are visiting new sites or producing new aborted subactions, unknown to the topaction at that time. Quiescing all the activity on behalf of the topaction *before* it commits can solve this problem, but the imminent cost of communication, roughly the same as needed for the Two Phase Commit protocol[1], voids such a solution. A better approach is to quiesce the remaining active (disconnected) actions and catch up on the missing information while carrying out the Two Phase Commit protocol, taking advantage of the existing message flow.

The current Two Phase Commit protocol [Liskov, *et al.* 1987b] has to be modified to take into consideration the (disconnected) actions that are still active and the fact that not all the participants are known when the protocol commences.

[1]It would take several rounds of messages from the coordinator to the known participants until all the participants are found and active DAs aborted.

The time-space diagram in Figure 5-1 depicts the modified Two Phase Commit protocol[2]. The details are given in Figure 5-2 for the coordinator and for the participants in Figures 5-3 and 5-4. The new protocol uses the first phase to quiesce active subactions and collect the missing names of participants and aborted (disconnected) actions. The second phase is similar to the second phase of the original, except that the list of missing aborted subactions has to be carried along.

The new protocol is as follows: The coordinator begins as before by sending a *PREPARE* message to all its known participants, accompanied by the accumulated list of known aborted subactions (*alist*). Each participant receiving the *PREPARE* message would also behave as in the original protocol, unless it participated in some disconnected subaction of the committing topaction.

A participant receiving a *PREPARE* message will do the following: Quiesce (i.e., abort) all the local active (disconnected) subactions of that topaction, including both top DAs and handler calls made by (descendants of) DAs. Prepare the local atomic objects (see Section 5.2) and forward the *PREPARE* message received from the coordinator to the participants of its committed local DAs. This way the message will get to **all** the participants. Some participants may get more than one *PREPARE* message, but they can simply ignore all but the first since the messages are identical.

All the participants that prepared successfully (see details in Section 5.2) reply with an *OK* message to the coordinator. A participant that is unable to prepare (e.g., because it crashed) replies with *REFUSED* as in the original protocol. The *OK* message is accompanied by two lists:

- $plist_d$ – the list of participants of committed DAs, created by combining the *plist*s of all the committed local top DAs.
- $alist_d$ – the list of aborted DAs, composed of those local top DAs that were aborted during preparation and all those in the *alist*s of the committed local top DAs.

The coordinator that receives *OK* messages from all the participants, including those in the

[2]See the original protocol in Figure 2-4 on page 30

```
% Simple version of the coordinator. No optimizations.
% Some straightforward type specs are missing.
coordinator = proc (topaction:action, alist: action_list, plist: site_list) returns (commit_or_abort)
        % Called by the topaction to perform Two Phase Commit
        % Begin phase I.
        for participant:site in site_list$elements(plist) do
                send_to(participant,"PREPARE",topaction,alist)
                end
        alist_rest : action_list := action_list$new()
        msg : message
        for participant:site in site_list$elements(plist) do
                msg := receive_from(participant)
                   except when timeout, aborted:
                        write_to_stable_storage("ABORT",topaction,plist)
                        for participant:site in site_list$elements(plist) do
                                send_to(participant,"ABORT",topaction)
                                end
                        return(abort)
                        end
†               site_list$add_to(plist,msg.plist_d)
                action_list$add_to(alist_rest,msg.alist_d)
                end
        % End of phase I. Coordinator decides to commit.
        write_to_stable_storage("COMMIT",topaction,plist,alist_rest)
        % Begin phase II.
        for participant:site in site_list$elements(plist) do
                send_to(participant,"COMMIT",topaction,alist_rest)
                end
        for participant:site in site_list$elements(plist) do receive_DONE(participant) end
        write_to_stable_storage("DONE",topaction)
        % End phase II.
        return(commit)
        end coordinator
```

Figure 5-2: Modified Two Phase Commit – The coordinator part

```
participant_phase_I = proc (topaction:action, alist:action_list, cc:crash_count)
        % Called at a site when a "PREPARE" message is received.
        % PREPARED  list of topactions that are locally prepared. (Atomic object !)
        % ACTIVE, COMMITTED  see Chapter 2 for detail.
        if action_list$is_in(topaction,PREPARED) then return end % Already received
"PREPARE"
        if cc > Local_Crash_Count then % Participant has crashed since visited
                send_to(topaction,"ABORT")
                return
                end
        action_list$insert(PREPARED,topaction)
        alist_d : action_list := action_list$new()
        plist_d : site_list := site_list$new()
        for active_action:action in active_list$elements(ACTIVE) do
                if aid$disconnected(active_action) cand
                        aid$descendant(active_action,topaction) then
                        % Found an active DA, abort it.
                        action_list$remove(ACTIVE,active_action)
                        if aid$top_DA(active_action) then
                                action_list$insert(alist_d,active_action)
                                end
                        end
                end
        for committed_action:action in committed_list$elements(COMMITTED) do
                if action_list$ancestor_is_in(alist,committed_action) then
                        committed_list$remove(COMMITTED,committed_action)
                        continue
                        end
                if aid$disconnected(committed_action) cand
                        aid$descendant(committed_action,topaction) then
                        % Found a committed top DA, get its alist and plist
                        action_list$add_to(alist_d,get_alist(COMMITTED,committed_action))
                        site_list$add_to(plist_d,get_plist(COMMITTED,committed_action))
                        end
                end
        prepare(topaction,alist) % see Section 5.2
        % Forward to participants of committed DAs
        for participant:site in site_list$elements(plist_d) do
                send_to(participant,"PREPARE",topaction,alist)
                end
        send_to(topaction,"OK",plist_d,alist_d)
        end participant_phase_I
```

Figure 5-3: Modified Two Phase Commit – The participant part, phase I

$plist_d$s[3] , can decide to commit the topaction. (In Chapter 6 we explore the possibility of doing so before all the participants have replied, or even if some refused.) The coordinator knows, at this point, about all the participants and all the aborted subactions. It has to process the lists returned with the *OK* replies: Add those new unknown participants to its *plist*, and merge the lists of aborted (disconnected) subactions (i.e. $alist_d$) into a new *alist*, $alist_{rest}$.

After making its decision, the coordinator can start the second commit phase. As in the original protocol, a *COMMIT* (or *ABORT*) message is sent to all the participants. Since the original *alist*, which was sent with the *PREPARE* message, lacked the aborted disconnected actions, the missing actions must be passed to the participants during the second phase. The participants that receive the *COMMIT* message with the new $alist_{rest}$ have at this point enough information to make the transaction's modifications permanent. They reply with *DONE* once done.

Note that the recursive case (i.e., nested DAs) is handled correctly by our scheme. Participants receiving forwarded *PREPARE* messages can forward them again. The *PREPARE* message would propagate to all the participants and they would all become known to the coordinator before it has to decide about the transaction's fate.

```
participant_phase_II = proc (fate:string, topaction:action, alist_rest:action_list)
        % Called at a site when a "COMMIT"/"ABORT" message is received.
        if fate = "COMMITTED" then
                commit_prepared(topaction,alist_rest) % see Section 5.2
                send_to(topaction,"DONE")
            else % fate = "ABORT"
                abort_prepared(topaction)          % see Section 5.2
                end
        action_list$remove(PREPARED,topaction)
        end participant_phase_II
```

Figure 5-4: Modified Two Phase Commit – The participant part, phase II

[3]Note the iteration semantics (in Figure 5-2, line marked with "†") : New sites from $plist_d$, added to *plist* in this iteration, would be yielded as participants in the following iterations.

5.1.1 Optimizations and variations

The modified Two Phase Commit protocol proposed above can be further optimized for better performance:

- The coordinator can add its *plist* to the *PREPARE* message. This way a participant may save on forwarding the message to some of the participants of the disconnected actions as they are already known by the coordinator, and return a smaller $plist_d$ to the coordinator.
- A participant may delay aborting some DAs after a *PREPARE* message is received to give them a chance to commit. The delay period has to be set empirically or be given by the programmer.
- A participant may add its $alist_d$ to the *alist* of the *PREPARE* message it forwards to the participants in its $plist_d$. This way those participants would have to write less to stable storage (see Section 5.2), but would have to monitor all the incoming *PREPARE* messages for different *alist*s.

5.2 Preparing an atomic object at a site

Preparing an atomic object that was modified by disconnected actions is different from the regular case: The object's site may not have enough information about the actions that modified the object during the first commit phase, and therefore it has to defer some decisions to the second phase.

An atomic object that was modified by an atomic transaction has an (stack) ordered list of potential versions. In a model without DAs, a participant receives enough information by the time it has to prepare to find the topmost version that belongs to a non-aborted descendant of the topaction. The participant prepares by writing that version to stable storage[4]; all the other versions can be discarded. The decision to abort or commit the topaction will determine which version will prevail – that top one or the base version.

[4] Writing a version to stable storage includes also non-stable objects that are accessible from this version. See [Oki, Liskov & Scheifler 1985] and [Kolodner 89].

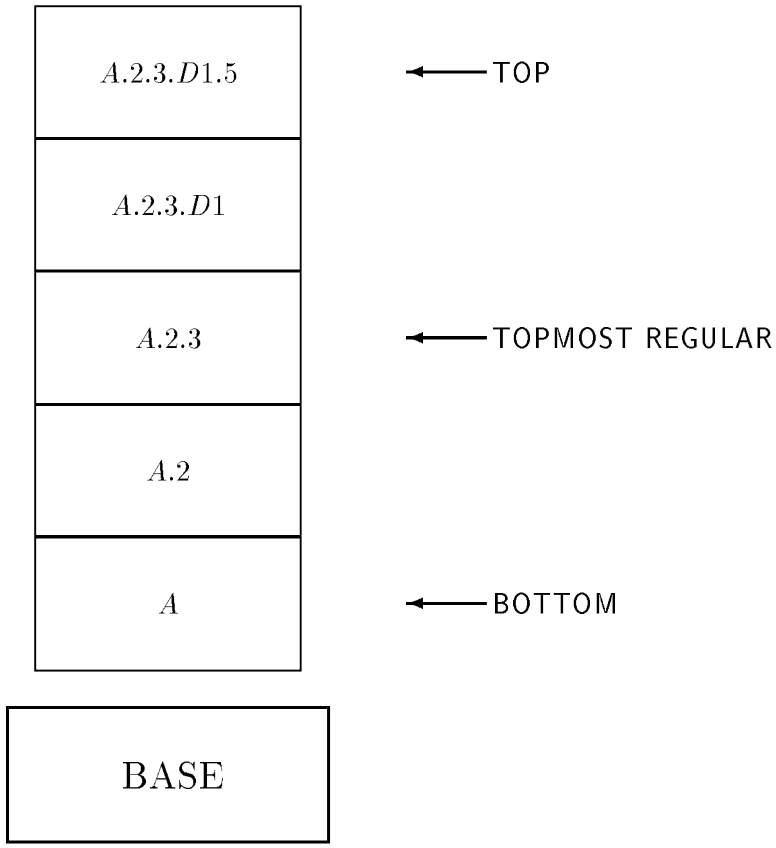


Figure 5-5: An example showing versions of an object to be prepared

When DAs are used and the above modified Two Phase Commit protocol is obeyed, a more complex scheme has to be used to ensure that the topmost committed version of an atomic object will be available when and if the topaction decides to commit. A participant may not have all the information it needs by the time it has to prepare to determine the topmost version of a committed subaction.

Figure 5-5 shows an example of an object at a site at the time when a *PREPARE* message was received. Note that since the locking rules enforce strict ancestry order among the versions, an object modified by DAs must have DAs' versions above the regular versions. Therefore if none of the actions holding versions on the object have an ancestor in the *PREPARE alist*, any of the topmost regular version or the DAs' versions could be the final one. In the example, if no ancestor of $A.2.3$ was in the *PREPARE alist*, the preparing participant can not tell which of $A.2.3$, $A.2.3.D1$ or $A.2.3.D1.5$'s versions will replace the base version if the coordinator decides

```
%
prepare = proc (topaction:action, alist:action_list)
        % called by participant_phase_I
        %
        prepared_objects : object_list := object_list$new()
        for obj:object in committed_list$all_objects(COMMITTED,topaction) do
                for ver:version in object$versions_top_to_bottom(obj) do
                        if action_list$ancestor_is_in(alist,ver.aid) then
                                object$discard_version(obj,ver)
                            elseif aid$not_disconnected(ver.aid) then
                                object$discard_versions_below_version(obj,ver)
                                break
                                end
                        end
                if object$have_no_versions(obj) then          % Read only object
                        object$release_all_locks(obj)
                    else
                        object_list$add_object(prepared_objects,obj)
                        % write remaining versions to stable storage
                        object$save_versions(obj)
                        end
        save_record("PREPARED",topaction,prepared_objects)
        end prepare
```

Figure 5-6: Preparing objects at the participant's site

to commit.

Preparing an object with DAs' versions requires writing all these versions to stable storage, in addition to the topmost regular version, unless some ancestor is known to be aborted. Figure 5-6 specifies the steps to be taken: When a *PREPARE* message is received, the versions on each of the topaction's object are scanned top to bottom. Versions of aborted subactions are discarded, and the remaining DAs' versions and the topmost regular version, if any exist, are written to stable storage. All the regular versions except the topmost one are discarded.

After all the topaction's objects have been prepared successfully, a record is written to stable storage to mark that fact. From this point on the site is committed to keeping all the prepared objects until notified about the coordinator's final decision.

The participant's work can be finished after the coordinator decision is received. As in the original protocol, when the decision is to abort the transaction, all the prepared versions and

```
%
abort_prepared = proc (topaction:action)
    % called by participant_phase_II
    %
    prepared_objects : object_list := get_record("PREPARED",topaction)
    for obj:object in object_list$elements(prepared_objects) do
        discard_prepared_material(obj)
        object$release_all_locks(obj)
        end
    end abort_prepared
```

Figure 5-7: Abort: Discarding prepared modifications at a participating site.

```
%
commit_prepared = proc (topaction:action, alist_rest:action_list)
    % called by participant_phase_II
    %
    prepared_objects : object_list := get_record("PREPARED",topaction)
    for obj:object in object_list$elements(prepared_objects) do
        for ver:version in object$versions_top_to_bottom(obj) do
            if action_list$ancestor_is_NOT_in(alist_rest,ver.aid) then
                object$replace_base_version_with_ver(obj,ver)
                break
                end
            end
        discard_prepared_material(obj)
        object$release_all_locks(obj)
        end
    end commit_prepared
```

Figure 5-8: Commit: Making prepared modifications permanent at a participating site.

locks are discarded (Figure 5-7). When the coordinator decides to commit, the participant has to finish the work from the first phase (i.e., identify versions of aborted subactions) before it can replace the base version with the correct new version. As detailed in Figure 5-8, the participant scans, for every object, the prepared versions top to bottom. A version found to belong to an aborted (disconnected) subaction is skipped; the first to belong to non-aborted subaction (if any) replaces the base version.

Chapter 6

Fault tolerant Two Phase Commit

In this chapter we explore the possibility of successfully committing a transaction in spite of some failures. That is, we want the Two Phase Commit coordinator to be able to *ignore* several participants that failed and commit the transaction without them.

The ability to ignore some participants gives more fault tolerance to our model. In the current Two Phase Commit protocol ([Gray 1978]), a failure of any participant either causes the transaction to abort or delays its commit.

The basic idea behind this chapter is that some *DAonly* participants, those that were used only by DAs[1], are dispensable. Since the effects created by Disconnected Actions are not essential to their creators' success, the transaction can commit without some of its DAonly participants.

A simple example is the replication application from Section 1.1, depicted in Figure 6-1. The topaction A ran at G_A and created five DAs ($A.D1, .., A.D5$) to update the object O that is replicated at $G_{D1}, .., G_{D5}$. After a majority of the DAs (i.e., three) committed, A continued running until it decided to commit. Meantime the remaining (two) DAs also committed to A. The transaction A had not modified any objects besides the replicas $O_1, .., O_5$.

The coordinator of A's Two Phase Commit can commit A even if one or two of the participants ($G_{D1}, .., G_{D5}$) failed to reply in time to the *PREPARE* message or replied with *REFUSED*! All the coordinator has to guarantee is that at least a majority of the participants did

[1]Note that these are exactly those participants of the $plist_d$s that are not in the coordinator's initial $plist$.

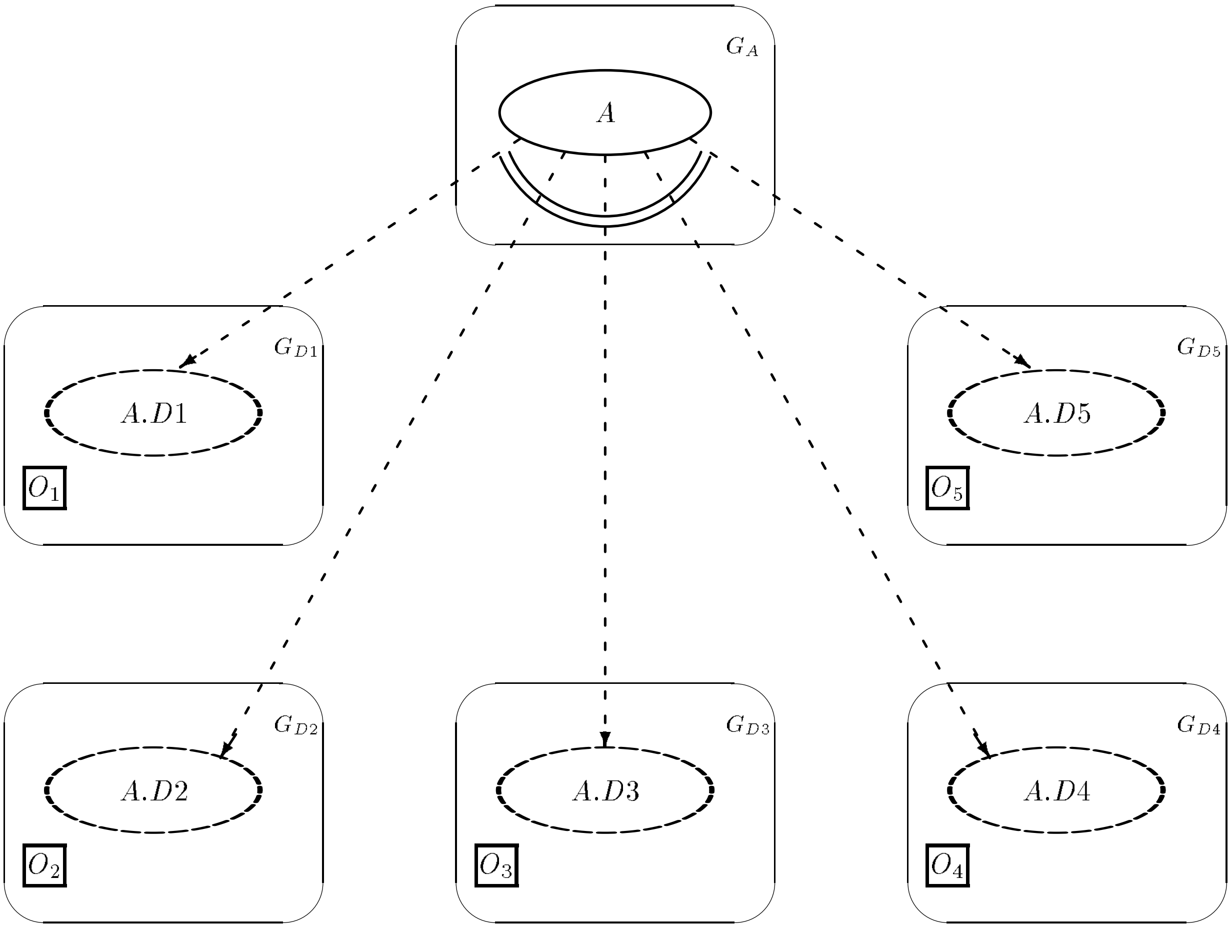


Figure 6-1: The replication example: Up to 2 sites can be ignored during Two Phase Commit.

manage to prepare, hence ensuring consistency of the replicated object.

The general case is not as straightforward as the previous example. In Section 6.1 we study the implications of ignoring DAonly participants. In Sections 6.2 and 6.3 we propose two solutions: introduction of a Disconnected Nested Topaction and modification of the existing model. In Section 6.4 we summarize and evaluate the merits of the ability to ignore participants.

6.1 Problems with Ignoring Disconnected Participants

Though in some special cases ignoring a participant has the effect of aborting a DA, the general case is far more complicated. Figure 6-2 demonstrates two possible problems: A DA affecting more than one site and a subaction depending on a DA.

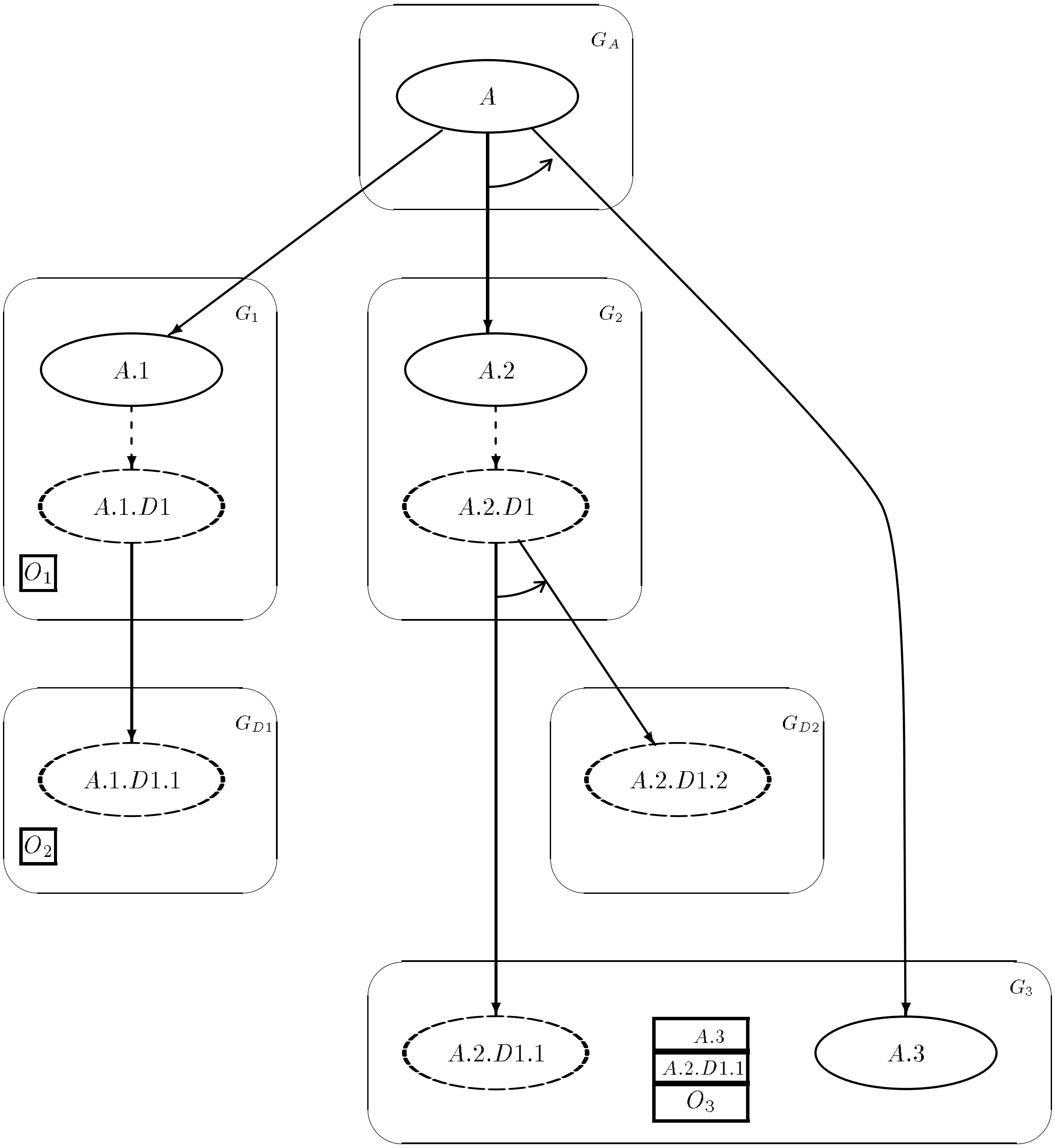


Figure 6-2: Problems with ignoring DAonly participants.

The problem of a DA creating effects at several participants is that ignoring one failed site is not equivalent to aborting the DAs that ran there. The problem is not trivial because the coordinator does not necessarily know which DAs ran at a certain site. Had it known, it could have added those DAs to its $alist_{rest}$, effectively aborting those DAs.

For example, assume that the DA $A.1.D1$ has modified two objects, O_1 at site G_1 and O_2 at G_{D1}, preserving some invariant between the two objects (e.g. value equality). An inconsistent state will be created if the coordinator ignores G_{D1} (e.g., because G_{D1} crashed) and decides to commit A, making A's effects permanent at all the other participants. Had the coordinator known that the DA $A.1.D1.1$ ran at G_{D1}, it could have aborted $A.1.D1.1$ by adding it to its $alist_{rest}$, preventing the problem of possible inconsistent state.

With the modified Two Phase Commit protocol, discussed in Chapter 5, the coordinator can not tell which DAs ran at a particular (DAonly) participant[2]. This information is distributed among the participants where the DAs were created, and is discarded by the end of the first commit phase.

The second problem concerns the dependency of other subactions on a commit of a DA. Aborting this DA would nullify these subactions and necessitate aborting them, which may not be possible when the dependent subactions are not disconnected.

For example, subaction $A.3$ in Figure 6-2 has read the object O_3 at site G_3, which was modified by $A.2.D1.1$ that committed. Ignoring G_{D2}, which implies aborting $A.2.D1$, would force the whole transaction to abort since the effects of $A.3$, a regular subaction, can not be undone. In the case where a subaction like $A.3$ is disconnected, cascading aborts can take place and allow the transaction to commit, but the coordinator has to know about the dependency.

6.2 First solution: Disconnected Nested Topactions

The two problems mentioned above can be avoided with a relatively simple mechanism: The *Disconnected nested topaction.* This new feature is basically a nested topaction that executes asynchronously with its creator, just like a DA, but unlike a nested topaction, the disconnected nested topaction (*DNTA*) can commit only if the topaction of its creator (i.e. the *main*

[2] Assume that, in our example, other DAs were also created at G_{D1}.

topaction) commits and none of its ancestors abort.

The DNTA solves the problem of other subactions that observe its effects by simply preventing it. Like a regular nested topaction, the DNTA is not considered a descendant of its creator for the purpose of lock inheritance or acquisition. The DNTA can not get a write-lock on an object locked by a descendant of the main topaction, or read an object that has a write-lock on behalf of the main topaction, and the same applies in the other direction.

The problem of a DA affecting several sites is solved by having centralized control over the whole DNTA at its coordinator. The DNTA's coordinator isolates the main coordinator from the DNTA's Two Phase Commit; it knows of all the DNTA's participants and when one has to be ignored, it can abort the whole DNTA.

6.2.1 Implementation of the DNTA

The Disconnected Nested Topaction starts as a DA, runs as a nested topaction and ends up like a DA again. This way a minimal change to the existing implementation is required. The DNTA is a nested topaction when needed to cope with the previous two problems, and is a DA for the purpose of committing it as part of the main topaction.

The implementation of the DNTA is as follows:

- The DNTA is created as a top DA[3], and is added to the list of active local actions.

- For lock propagation purposes the DNTA is considered a nested topaction.

- The nested topaction of the DNTA runs and commits as usual until it finishes phase one of its Two Phase Commit (see Figure 5-2). If its coordinator decided to commit, it does not write to stable storage or send any messages, but instead commits like a top DA and keeps (in its *COMMITTED* entry) its *plist* (and its $alist_{rest}$ if it had nested DAs); there is no need to keep the list of objects since they are all prepared at their sites (those in the *plist*).

  Note that getting to this point means that none of the DNTA's participants aborted, and they are all at the "prepared" state. Had any participant failed, the coordinator of the

[3]The DTNA would need a special tag in its AID if DTNAs and DAs are to coexist in the system.

nested topaction would have had to abort it, notify its participants and remove all traces of it from the local site. Note that a crash of the local site (before preparing for the main topaction) would have the same effect as aborting except that messages are not sent to the participants. In this case the participants would have to initiate queries to find that their DNTA aborted.

- The modified Two Phase Commit (described in Chapter 5) of the main topaction would finish up the DNTAs' work correctly. In the prepare phase, those DNTAs with aborted ancestors would be aborted (see Figure 5-3[4]), and the *plists* (and *alists* in the recursive case) of the committed DNTAs would be added to the local $plist_d$ (and $alist_d$). This way the second phase of the DNTA's commit would be handled by the second commit phase of the main topaction.

  A small semantic change is needed, though, in the second commit phase. The coordinator's decision message should apply not only to its transaction but to its (separately kept) DNTAs at the participants. The algorithms in Chapter 5 need not be changed, except that the "get_record" operation, used in Figures 5-8 and 5-7, should be specified to also retrieve the records of the transaction's DNTAs.

6.2.2 Evaluation of DNTAs

The main advantage of the Disconnected Nested Topaction is its simple implementation. As seen above, only three minor changes have to be made to our model in order to have DNTAs in it.

The main disadvantage of DNTAs is their restricted functionality. They can not inherit (or release) locks from (or to) their creators. Programmers have to be very careful to avoid deadlocks with the use of DNTAs. For example, a transaction trying to read an object that was modified by one of its DNTAs would wait forever.

Quorum-sets can be done with DNTAs instead of DAs. They would take longer to run because a prepare phase is involved, but any committed DNTA is guaranteed to survive crashes of its participants (except its coordinator's site).

[4] At that stage we can add sending abort messages to the participants of the aborted DNTAs.

$$
\begin{array}{ll}
2/3 \left\{ \begin{array}{ll} DA_1 & \{participant_1, participant_5\} \\ DA_2 & \{participant_1, participant_6\} \\ DA_3 & \{participant_1, participant_7\} \end{array} \right. \\
1 \quad\;\; \begin{array}{ll} DA_4 & \{participant_2, participant_3, participant_4\} \end{array} \\
3/5 \left\{ \begin{array}{ll} DA_5 & \{participant_3, participant_5, participant_1\} \\ DA_6 & \{participant_4\} \\ DA_7 & \{participant_5, participant_7\} \\ DA_8 & \{participant_6, participant_9, participant_{11}, participant_{12}\} \\ DA_9 & \{participant_7, participant_{10}\} \end{array} \right.
\end{array}
$$

Figure 6-3: An abstract view of a possible *maplist*

Note that, though restricting lock exchange, the DNTA releases its read-locks after finishing its first commit phase, therefore enabling its main topaction to get write locks on objects that were only read by DNTAs.

6.3 Second solution: Modify the commit mechanism

Site ignoring can also be achieved by some modifications to the commit mechanism. The modified mechanism enables the coordinator to deduce which DAs ran at a certain site and on which (DAonly) sites committed regular subactions depend.

To provide the coordinator with the mapping from a site to the names of the DAs that ran there, the participants have to send this information to the coordinator. Each participant has the participants lists for the committed top DAs. I.e., the mapping F:

$$\forall\ DA \in COMMITTED,\ \ F : DA \longrightarrow \{participant_1, participant_2, \ldots\}$$

By combining all the maps it receives, the coordinator has the inverse mapping F^{-1}:

$$\forall\ Participant \in plist,\ \ F^{-1} :\ Participant \longrightarrow \{DA_1, DA_2, \ldots\}$$

and can abort all the DAs that ran at the sites it ignored by adding them to its $alist_{rest}$.

The implementation is simple; each participant has to replace its $plist_d$, returned with its *OK* reply, with a *maplist*, a list of committed local top DAs, each DA with its *plist*. An example for an abstract *maplist* is given in Figure 6-3. The size of the *maplist* will be proportional to the number of DAs, but can be reduced if we use the optimization of adding the coordinator *plist* to the *PREPARE* message; a DA that has **all** its participants in the *PREPARE plist* need not be reported since none of its participants can be ignored.

Another problem to handle is DAs of quorum-sets. Suppose some action created a set of n DAs, required that at least q commit and m eventually did ($q \leq m \leq n$). The coordinator should not be allowed to violate this requirement. The participant should pass this information in the *maplist* with its *OK* reply, and the coordinator should check that the minimum for each set is not violated by ignoring some participants[5]. The example in Figure 6-3 shows two such sets; at least two DAs out of $\{DA_1, DA_2, DA_3\}$ and three out of $\{DA_5, \ldots, DA_9\}$ must commit.

Another way to abort DAs that ran at an ignored site is by letting the participants abort DAs. The coordinator will decide on ignoring a participant and tell the other participants about it. Each participant will use its local mapping to deduce which DAs should be aborted, and notify the sites in their *plist*s. This way less information has to be sent to the coordinator, but handling DAs of quorum sets would be more complicated.

The problem of a regular subaction S that depends on the commit of a DA D can be solved by letting the coordinator know about S's dependency on D's DAonly participants, hence preventing the coordinator from ignoring DAonly participants upon which committed regular subactions depend. This knowledge can be easily obtained from the mechanism that detects Crash Orphans ([Liskov, *et al.* 1987a]). If such detection mechanism is not implemented, a simpler mechanism can be devised (see Section 6.3.1) to acquire the dependency information.

The Crash Orphans mechanism works by maintaining, for every action, a list of sites on which the action depends (*dlist*). Besides regular sites, the *dlist* contains (possibly DAonly) sites that were used by DAs upon whom the action depends, as described in Section 3.4. In the above example, when $A.3$ tries to get the lock on O_3, the query to G_2 regarding $A.2.D1$'s fate would return with $A.2.D1$'s *dlist* (if it committed), which would be merged to $A.3$'s *dlist*. If (some ancestor of) $A.3$ aborted, the *dlist* would be discarded and not propagated up to the topaction.

All that has to be done to prevent the dependency problem is to give the topaction's *dlist* to the coordinator. The coordinator is not allowed to ignore any member of this *dlist*. This

[5]The participant can find this information in the entry of the concurrent-set action in *COMMITTED* (see Section 3.2.2)

solution is correct because all the DAonly sites upon which a regular action depends must be in its *dlist*, and when that action commits those sites are inserted into its ancestor's *dlist*. (Note that *dlist*s of aborted subactions are discarded.) To reduce the expected size of the *maplist*s that accompany *OK* replies, the coordinator can add the topaction's *dlist* to its *PREPARE* message (replacing the *plist* in the optimization mentioned above).

6.3.1 A new dependency detection mechanism

If the system does not use the Crash Orphan Detection mechanism, a smaller mechanism can be devised to serve our need. The new dependency detection mechanism is basically a subset of the Crash Orphan Detection mechanism, though much simpler because it does not need to catch crash orphans "on the fly", hence maps of crashcount need not be maintained and sent with every message. Another simplification comes from not using *dlist*s; we need not keep a total dependency relation, only dependency on sites used by DAs, and can use the actions' *plist*s to convey this information to the coordinator.

The idea behind the new mechanism is that when any action A acquires a lock on an object O, where committed DAs had versions (i.e., write locks), the *plist*s of those DAs are to be merged into A's *plist*. This way the coordinator would end up with some DAonly participants in it initial *plist*, which are those DAonly participants upon which committed regular subactions depend, and would not consider them as DAonly (i.e., ignorable).

This new use of the *plist* may change its semantics a bit, since it may contain sites that were not visited by its action's descendants, but will cause no problem because those semantics are only required from the $plist_{final}$, the *plist* that the coordinator has by the time it has to make the commit decision, and the following invariant holds (for a topaction T):

$$\forall action \in Descendants(T) : plist(action) \subseteq newplist(action) \subseteq dlist(action) \subseteq plist_{final}(T)$$

That is, the new *plist* may have more sites than the original, but no more than the *dlist* that is used when the Crash Orphan Detection mechanism is used, and no foreign sites are introduced into the coordinator's *plist*.

For completeness, the details of the new mechanism are given below (with references to the example in Figure 6-2 for clarity). Note that this is basically the same algorithm as was given in Section 3.4, only the *plist* is used instead of the *dlist*.

- When a query about a committed top DA is replied with "committed", the DA's *plist* accompanies the reply. (Such a query must be sent from G_3 to G_2 before $A.3$ can have a conflicting lock on O_3.)

- An (initially empty) *plist* is associated with every object version.

- A site receiving a query reply with a DA's *plist* would do the lock propagation as usual, but when a version is being inherited (e.g. $A.2.D1.1$'s $\longrightarrow A$), the *plist* received is merged into the version's *plist*. If the inheritor (A) already had a version, the other two *plist*s ($A.2.D1.1$'s and the one received) are merged into its (A's) *plist*.

- When an action gets a (read or write) lock on an object, all the *plist*s on the version read are merged into the action's *plist*.

6.3.2 Evaluation

Unlike the DNTA solution, the one above is transparent to programmers; it puts no programming restrictions. When fault tolerance during the Two Phase Commit process is important, dependencies among subactions (such as a regular subaction reading the effects of many DAs) should be avoided.

This solution also improves the performance of the Two Phase Commit somewhat by making more participants known to the coordinator before it begins the first phase.

The disadvantage of this solution is its performance cost. The maps that are sent with the *OK* reply may be large, and the Orphan Detection mechanism is expensive (see [Nguyen 1988]). Without Orphan Detection, our new dependency detection mechanism, which also has some cost, has to be implemented.

6.4 Summary

In this chapter we presented the idea of completing the Two Phase Commit process successfully in spite of failures. This new ability is derived from the semantics of disconnected actions, namely doing work that is not necessary for the correctness of their creators.

In applications like replication using majorities, the semantics of a DA may change as it commits and becomes part of the quorum, thus doing essential work like any other regular subaction. Nevertheless, its old semantics can be applied again during the Two Phase Commit process to tolerate failures if more than the minimal quorum has eventually committed.

The ability to ignore several participants can be very useful in large scale distributed systems. A system like *The ClearingHouse* ([Oppen & Dalal 1981]) maintains a naming database over more than a thousand sites, and the likelihood of a failure occurring during the lifetime of a transaction can not be overlooked. The ClearingHouse solved this problem by not using atomic transactions. With the scheme presented here, atomic transactions can be used in large scale distributed systems.

Another benefit from the new ability is shorter timeout periods. In the original model, timeouts were used by the Two Phase Commit protocol to decide that aborting the transaction (and possibly retrying) is more practical than waiting for a (failed) participant. Those timeout periods had to take into account the worst possible case: Longest message round trip time, longest processing time and longest disk write time. Practical timeout periods are order of magnitudes longer than the average case. With the ability to ignore participants, shorter timeout periods can be used as a failed participant does not necessarily means an aborted transaction.

A point that was hardly mentioned in the above chapter is recursively created DAs. It seem to us that all the work presented here can be generalized in a straightforward manner to include nested DAs; DTNAs can be nested, and in the second solution, the coordinator's handling of nested DAs is similar to handling non-nested ones.

Two possible implementations were proposed: Disconnected Nested Topaction and a modified commit mechanism. The trade-off is between simplicity of implementation and functionality. We tend to prefer the second solution as we do not trust the skills of programmers, who can easily misuse DNTAs. The second solution is also not difficult to implement when the system uses the Crash Orphan Detection mechanism.

Chapter 7

Conclusion

Below we summarize our work and propose directions for further research.

7.1 Summary

The ability to perform work on behalf of a nested action in parallel with its transaction can enhance a nested atomic action system to better exploit potential parallelism. The *disconnected action*, proposed in this thesis, provides a methodical way to perform such an activity.

Disconnected actions (DAs) can be used to perform benevolent side-effects within the action. Unlike a nested topaction, which creates side-effects independent of its creator, modifications made by DAs live within the scope of their creator; when an action aborts, all the modifications made by its disconnected (and regular) descendants are undone. Also, unlike a nested topaction, the DA can observe effects that were created by its transaction (prior to the creation on that DA) and execute in parallel with its transaction.

We described the current model of computation that is used by the Argus system; this model was the basis to our work. We made some small changes to the current model that helped us reason about our work. We separated the implementation of locks into an unordered set of read-locks and a stack of write-locks (versions) ordered in ancestry order, we redesigned the structure of the action identifier (AID) to reflect accurately the action's position in its transaction's action-tree, we introduced the concurrent-set action to help with problems special to concurrent subactions, and we stated clearly the way dependency lists (*dlists*) should be handled for orphan detection.

In Chapter 3 we described the role, use and implementation of DAs in our new model. We showed how the lock query mechanism and the orphan detection mechanism can be modified to work correctly when DAs are used.

We proposed, in Chapter 4, a technique to serialize disconnected actions. For serial DAs, we used timestamps, carried with each action and left on every lock, to tell a DA which effects were created prior to its creation and which are not; in this way we prevent non-serializable executions. Timestamps can not serialize DAs that were created by concurrent subactions, however, since they get serialized dynamically; therefore we introduced a new mechanism of conflict tables (CTs) for them. The CT mechanism keeps a table at the concurrent-set action's site to monitor the serialization of the concurrent subactions explicitly. We also introduced fingerprint locks to help retain access information at the object.

Our serialization technique is an ad hoc serialization method, a hybrid of timestamping-based protocol ([Reed 1978, Reed 1983]), locking-based protocol for nested atomic actions ([Moss 1981]) and an explicit serialization control method.

In Chapter 5 we described in detail how the two-phase commit protocol, which is used by the current model to commit the top-level action, is modified to cope with problems introduced by DAs: DAs may still be active when their topaction commits, and the sites they used and the subactions they aborted may not be known to the protocol's coordinator when it begins.

In Chapter 6 we introduced a novel idea of committing transactions in spite of some failures. The idea derives from the semantics of DAs; DAs basically do functionally redundant work. We proposed two ways in which some sites that were used only by DAs can be ignored, if necessary, during the commit process.

We feel that we achieved our design goals: minimizing the change to the design of the current model and its performance, and ensuring that disconnected actions have a fair chance of success. When DAs are not used, the new model does not send any extra messages, does very little additional local processing, and requires little additional storage; when DAs are used, they are almost as likely to succeed as regular subactions.

7.2 Future work

Below we present several directions in which more work can be done:

7.2.1 Use of an Explicit Approach to provide Atomicity

Though disconnected actions were initially introduced into the current model in order to perform functionally redundant work, they can also be used to do work that is essential to their transaction. One example is their use for quorum-sets, which is described in the thesis. Another potential "non-redundant" use of DAs can be achieved if the model is enhanced to provide programmers with explicit tools for implementing atomic types, as described in [Weihl 1984].

As an example for the use of the explicit approach, consider an operation OP on some abstract object that is implemented, as described earlier in this thesis, as two operations: the first (OP_1) does the real work (reads or modifies), and the second (OP_{DA}) runs as a DA and improves the representation of the object. As DAs were described in our model, they are not guaranteed to succeed; therefore a subsequent use of OP_1 may find that the previous OP_{DA} is yet not done, and OP_1 would do the necessary work itself, probably causing OP_{DA} to abort.

The implementation can be improved if, using the explicit approach, the second use of OP_1 could tell that the prior OP_{DA} is not yet done, thus requiring a delay of the second OP_1 for a short period until the first OP_{DA} commits. This would result in a better performance than of aborting the first OP_{DA} and starting its work all over again.

7.2.2 Use a Reed-like method for serial DAs

Our use of timestamps for serializing DAs was only a simple addition to the existing locking-based mechanism; DAs found to access the object out of order were aborted, except for the case of multiple readers. In Reed's scheme ([Reed 1978]), many versions are kept for an object, ordered in their timestamps order; thus the conflicts of multiple writers, or a reader that reads after a later writer wrote, can be sorted out without aborting by accessing an earlier version, rather than the most recent one. A small improvement to Reed's scheme is proposed in [Aspnes, *et al.* 1988]; it keeps all the read-timestamps, not only the maximal as does Reed, and subsequent aborts allow for a reduction in the value of the maximal read-timestamp.

We can have a different scheme for serial DAs, which is similar to Reed's. Note that our locks (and versions) resemble Reed's multiple-version scheme. It seems that the cases of multiple writers and a late reader could be sometimes solved without the need to abort.

7.2.3 Use other Deadlock Detection methods for Concurrent DAs

As stated before, we find the problem of deadlock detection and serialization analogous; a cycle in the wait-for graph means a deadlock, and a cycle in our serialized-after conflict table means non-serializable execution.

Most deadlock detection methods for distributed systems build some variation of a wait-for graph in either a *centralized* or *distributed* manner. The centralized ones (including hierarchical methods, see [Menasce & Muntz 1979]) resemble our constraint table method; they use a distinguished node in which the wait-for graph is built. The distributed protocols (e.g., [Menasce & Muntz 1979, Moss 1981, Obermarck 1982]) create the wait-for graph, whole or in part, at some node where a local transaction suspects that it is deadlocked. The distributed protocols are also known as *edge-chasing* ones because they try to detect a cycle by following the graph's edges, often requiring messages to be sent between nodes that have transactions along the wait-for path.

We have chosen a centralized method to prevent non-serializable execution of concurrent subactions that use DAs because it was simpler and seemed to require less additional messages. We believe that a distributed method, similar to some distributed deadlock detection protocol, can also be devised.

7.2.4 Optimistic Model

Work can be done on using disconnected actions in a model that uses optimistic concurrency control, like the one in [Gruber 1989]. The optimistic approach allows atomic action to execute without synchronization (e.g., block when another action uses a needed object), and rely on commit-time validation to ensure serialization of the actions. Optimistic methods are useful when things are not likely to "go wrong", since they pay a higher penalty (than pessimistic ones) if things do go wrong.

A quasi-optimistic[1] method can be combined with our pessimistic one to handle the case of quorum-sets. Our technique of starting up some N DAs concurrently, while only a quorum of M ($M < N$) needs to commit, has a high success probability. This technique can be speeded up with an optimistic model, since it is unlikely to fail; the creator of the quorum-set need not be blocked until the quorum commits, but can continue, and be undone if less than the quorum committed.

7.2.5 Claiming DAs

DAs, as presented in our work, are synchronized with their transaction only when the latter commits. Therefore DAs can not return replies to their transaction like handler calls do. A linguistic support for asynchronous work was proposed in [Liskov & Shrira 1988] with the introduction of *promises* into the language. A promise is returned to the caller when an asynchronous activity begins, and can be used by the program to reclaim the results of that activity later.

More work can be done to try and combine the two methods; a promise can be supported by our model by creating a DA to perform the asynchronous activity, and maintaining enough information in the implementation of the promise (e.g., the DA's AID) to track the relevant DA and find out about its fate and its promised reply.

7.2.6 Other Ideas

Some work can be done to finish up working out the details of the points we ignored in the description of the conflict table mechanism: the handling of aborts, discarding of obsolete read FP locks and not delaying regular actions. The details of extending our algorithms and protocols to handle nested DAs need to be filled in. Similar to the extension of timestamping, the rest of the work is likely to require only straightforward extension.

[1]This method is not fully optimistic because locking is still used.

Appendix A

New implementation of the AID

The following possible implementation of an AID is compatible with its use in this thesis:

```
% Structure:
%        AID     > NEW_GUARD<num>TOP_ACTION<num><body>
%        <body> > <aid_tag><num><body> | epsilon
%
aid = cluster is new, make_subaction, make_nestedtop, make_hcall, make_DA, LCA,
      make_concurrent_set_action, descendant, non_descendant, proper_descendant,
      not_concurrent_relatives, disconnected, not_disconnected, topaction, top_DA,
      concurrent_set_action, hcall, get_parent, get_location, get_topaction,
      subaction, concurrent_subactions_between, locations_of_disconnected_ancestors,
      lt, gt, equal, copy

%%
%%   Denitions
%%

%% Tags for the levels in an AID

aid_tag =
oneof[TOP_ACTION,SUB_ACTION,NEW_GUARD,CONC_SET_ACTION,DISC_ACTION:null]
TOP_ACTION = aid_tag$make_TOP_ACTION(nil)
SUB_ACTION = aid_tag$make_SUB_ACTION(nil)
NEW_GUARD = aid_tag$make_NEW_GUARD(nil)
CONC_SET_ACTION = aid_tag$make_CONC_SET_ACTION(nil)
DISC_ACTION = aid_tag$make_DISC_ACTION(nil)

aid_level = struct[tg : aid_tag, nm : int]
site = int

rep = sequence[aid_level]

own TopActionID : int := 0        %%% ID generator for TopActions
```

```
%%
%%      Make AIDs
%%

new = proc (GID:int) returns (cvt)
      % RETURNS an aid for a non-nested top action at guardian # GID
      TopActionID := TopActionID + 1          % increment ID generator
      return(rep$[aid_level${tg: NEW_GUARD, nm: GID},
                  aid_level${tg:TOP_ACTION, nm:TopActionID}])
      end new

make_subaction = proc (Parent: cvt, Step: int) returns (cvt)
       % Requires: Step bigger than the Step given previously by the same action
       % Eects: Returns an aid for a local sub action as the Parent's child
       %        (Enumerated with Step)
       return(rep$addh(Parent,aid_level${tg:SUB_ACTION, nm:step}))
       end make_subaction

make_concurrent_set_action = proc (Parent: cvt, Step: int) returns (cvt)
       % Requires: Step bigger than the Step given previously by the same action
       % Eects: Returns an aid for a concurrent-set action as a child of Parent
       return(rep$addh(Parent,aid_level${tg:CONC_SET_ACTION, nm:step}))
       end make_concurrent_set_action

make_nestedtop = proc (Parent: cvt, Step: int) returns (cvt)
       % Requires: Step bigger than the Step given previously by the same action
       % Eects: Returns an aid for a nested top action as a child of Parent
       return(rep$addh(Parent,aid_level${tg:TOP_ACTION, nm:step}))
       end make_nestedtop

make_DA = proc (Parent: cvt, step: int) returns (cvt)
       % Requires: Step bigger than the Step given previously by the same action
       % Eects: Returns an aid for a top disconnected action as a child of Parent
       return(rep$addh(Parent,aid_level${tg:DISC_ACTION, nm:step}))
       end make_DA

make_hcall = proc (Parent: cvt, GID: int) returns (cvt)
       % Requires: Step bigger than the Step given previously by the same action
       % Requires: Parent is a call-action
       % Eects: Returns an aid for a handler call (at site GID) as a child of Parent
      return(rep$addh(Parent,aid_level${tg:NEW_GUARD, nm:GID}))
      end make_hcall
```

```
%%
%% Check relations between AIDs
%%

any_descendant = proc (Child, Parent: rep) returns (bool)
        % Eects: Returns true i Child is a descendant of Parent (or Child = Parent)
        if rep$size(Parent) > rep$size(Child) then return(false) end
        possible_parent : rep := rep$subseq(Child,1,rep$size(Parent))
        return(rep$similar(possible_parent,Parent))
        end any_descendant

nested_topaction_between = proc (Child, Parent: rep) returns (bool)
        % Requires: Child is a descendant of Parent
        % Eects: returns true i Child belongs to some nested topaction that
        %         was created by a descendant of Parent.
        for index:int in int$from_to(rep$size(Parent)+1,rep$size(Child)) do
                if Child[index].tg = TOP_ACTION then return(true) end
                end
        return(false)
        end nested_topaction_between

descendant = proc (Child, Parent: cvt) returns (bool)
        % Eects: Returns true i Child is a descendant of Parent. However, if
        %         there is a nested topaction that is an ancestor of Child and a
        %         proper descendant of Parent, returns false.
        return(any_descendant(Child,Parent) cand ~nested_topaction_between(Child,Parent))
        end descendant

non_descendant = proc (Child, Parent: cvt) returns (bool)
        % Eects: Returns false i Child is a descendant of Parent. However, if
        %         there is a nested topaction that is an ancestor of Child and a
        %         proper descendant of Parent, returns true.
        return(~descendant(up(Child),up(Parent)))
        end non_descendant

proper_descendant = proc (Child, Parent: cvt) returns (bool)
        % Eects: Returns true i Child is a proper descendant of Parent.
        %         (And there is no nested topaction between the two).
        return(descendant(up(Child),up(Parent)) cand ~rep$similar(Child,Parent))
        end proper_descendant

not_concurrent_relatives = proc (A1,A2:cvt) returns (bool)
        % Eects: Returns true i A1 and A2 are not concurrent relatives
        the_LCA : rep := down(LCA(up(A1),up(A2)))
           except when not_related: return(true) end
        return( rep$top(the_LCA).tg ~= CONC_SET_ACTION )
        end not_concurrent_relatives
```

```
%%
%% Check the type of an AID
%%

disconnected = proc (A:cvt) returns (bool)
    % Eects: Returns true i A is disconnected (i.e., descendant of top DA)
    for level: aid_level in rep$elements(A) do
        if level.tg = DISC_ACTION then return(true) end
        end
    return(false)
    end disconnected

not_disconnected = proc (A:cvt) returns (bool)
    % Eects: Returns false i A is disconnected (i.e., descendant of top DA)
    return(~disconnected(up(A)))
    end not_disconnected

topaction = proc (A: cvt) returns (bool)
    % Eects: Returns true i A is a [nested] top action
    return(rep$top(A).tg = TOP_ACTION)
    end topaction

top_DA = proc (A: cvt) returns (bool)
    % Eects: Returns true i A is a top disconnected action
    return(rep$top(A).tg = DISC_ACTION)
    end top_DA

concurrent_set_action = proc (A:cvt) returns(bool)
   % Eects: Returns true i A is a concurrent-set action
   return(rep$top(A).tg = CONC_SET_ACTION)
   end concurrent_set_action

subaction = proc (A:cvt) returns(bool)
   % Eects: Returns true if A is a regular subaction (and nothing else)
   return(rep$top(A).tg = SUB_ACTION)
   end subaction

hcall = proc (A:cvt) returns(bool)
    % Eects: Returns true i A is a handler call
    return(rep$top(A).tg = NEW_GUARD)
    end hcall
```

```
%%
%%      Miscellaneous
%%

concurrent_subactions_between = iter (Parent, Child:cvt) yields (cvt)
        % Requires: Parent is an ancestor of Child
        % Eects: Yields every child A of a concurrent-set action such that A
        %        is an ancestor of Child and a proper descendant of Parent
        for index: int in int$from_to(rep$size(Parent)+1,rep$size(Child)) do
                if Child[index-1].tg = CONC_SET_ACTION then
                        yield(rep$subseq(Child,1,index))
                        end
                end
        end concurrent_subactions_between

get_parent = proc (A:cvt) returns(cvt)
        % Eects: Returns A's parent (or A if A is a (non-nested) topaction)
        if rep$size(A) = 2 then return(A) end
        return(rep$remh(A))
        end get_parent

LCA = proc(A1, A2:cvt) returns (cvt) signals (not_related)
        % Eects: Returns the AID of LCA(A1,A2), signals not_related if needed
        if A1[1] ~= A2[1] cor A1[2] ~= A2[2] then signal not_related end
        short : rep
        if rep$size(A1) > rep$size(A2) then short := A2 else short := A1 end
        for index:int in int$from_to(2,rep$size(short)) do
                if A1[index] ~= A2[index] then return(rep$subseq(A1,1,index-1)) end
                end
        return(short)
        end LCA

get_location = proc (A: cvt) returns (site)
        % Eects: Returns the site of action A
        for index:int in int$from_to_by(rep$size(A),1,-1) do
                if A[index].tg = NEW_GUARD then return(A[index].nm) end
                end % for
        end get_location

get_topaction = proc (A: cvt) returns (cvt)
        % Eects: returns the immediate (possibly nested) topaction of A
        for index:int in int$from_to(rep$size(A),2) do
                if A[index].tg = TOP_ACTION then return(rep$subseq(A,1,index)) end
                end
        end get_topaction
```

```
locations_of_disconnected_ancestors = iter(A: cvt) yields (site,cvt)
        % Eects: Yields pairs of top disconnected ancestors of A and their sites
        for index: int in int$from_to(1,rep$size(A)) do
                if A[index].tg = DISC_ACTION then
                        temp_aid : rep := rep$subseq(A,1,index)
                        yield(get_location(up(temp_aid)),temp_aid)
                        end
                end
        end locations_of_disconnected_ancestors

equal = proc (A1, A2: cvt) returns (bool)
        % Eects: Returns true i A1 and A2 denote the same action
        return( A1 = A2 )
        end equal

lt = proc (A1, A2: cvt) returns (bool)
        % Requires: A1 and A2 belong to same transaction
        % Eects: Creates a total order based on AIDs enumeration. (A1 < A2)
        %        (Returns true if A1 was created before A2, meaningless if A1 and
        %          A2 are concurrent siblings)
        return(gt(up(A2),up(A1)))
        end lt

gt = proc (A1, A2: cvt) returns (bool)
        % Requires: A1 and A2 belong to same transaction
        % Eects: Creates a total order based on AIDs enumeration. (A1 > A2)
        %        (Returns true if A1 was created after A2, meaningless if A1 and
        %          A2 are concurrent siblings)
        point_of_di : int := rep$size(down(LCA(up(A1),up(A2)))) + 1
        min_size : int := int$min(rep$size(A1),rep$size(A2))
        if point_of_di = min_size + 1 then
                return( rep$size(A1) > min_size )
            else
                return( A1[point_of_di].nm > A2[point_of_di].nm )
                end
        end gt

copy = proc (A:cvt) returns (cvt)
        return(rep$copy(A))
        end copy

end aid
```

Appendix B

Implementation of Timestamps

The following possible implementation of a timestamp is compatible with its use in this thesis:

```
% Structure:
%           timestamp -> <num>timestamp | <num>
%
timestamp = cluster is new, increment, nest, equal, similar, lt, gt, max

     rep = sequence[int]

     new = proc() returns(cvt)
          % Eects: Create and return a new (zero) timestamp
          return(rep$[0])
          end new

     increment = proc (t:cvt) returns(cvt)
          % Eects: Increments the current level by 1
          return(rep$addh(rep$remh(t),rep$top(t) + 1))
          end increment

     nest = proc (t:cvt) returns(cvt)
          % Eects: Adds a new level, initially 0
          return(rep$addh(t,0))
          end nest

     equal = proc (t1,t2:cvt) returns (bool)
          % Eects: Returns true i two timestamps are one
          return(t1 = t2)
          end equal

     similar = proc (t1,t2:cvt) returns (bool)
          % Eects: Returns true i two timestamps have the same value
          return(rep$similar(t1,t2))
          end similar
```

```
lt = proc (t1,t2:cvt) returns (bool)
        % Eects: Returns true i t1 < t2
        for index:int in rep$indexes(t1) do
                if t1[index] > t2[index] then return(false)
                   elseif t1[index] < t2[index] then return(true)
                      end except when bounds: return(false) end
                end % for
        return(false)
        end lt

gt = proc (t1,t2:cvt) returns (bool)
        % Eects: returns true i t1 > t2
        return(~lt(Rp(t1),up(t2)) cand ~equal(up(t1),up(t2)))
        end gt

max = proc (t1,t2:cvt) returns(cvt)
        % Eects: Returns the larger of two timestamps
        if lt(up(t1),up(t2)) then return(t2) else return(t1) end
        end max

end timestamp
```

Bibliography

[Allchin & McKendry 1983] J. E. Allchin and M. S. McKendry. Synchronization and Recovery of Actions. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, Montreal, Canada*, pages 31–44, ACM, August 1983.

[Aspnes, *et al.* 1988] James Aspnes, Alan Fekete, Nancy Lynch, Michael Merritt, and William Weihl. A Theory of Timestamp-Based Concurrency Control for Nested Transactions. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 431–444, August 1988.

[Bernstein & Goodman 1981] Philip A. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[Bjork 1973] L. A. Bjork. Recovery Scenario for a DB/DC System. In *Proceedings of the ACM Annual Conference*, pages 142–146, ACM, Atlanta, GA, 1973.

[Boggs, *et al.* 1979] David R. Boggs, John F. Shoch, Edward A. Taft, and Robert M. Metcalfe. *Pup: An Internetwork Architecture*. CSL 79-10, Xerox Palo-Alto Research Center, July 1979.

[Davies 1973] C. Davies. Recovery Semantics for a DB/DC System. In *Proceedings of the ACM National Conference 28*, pages 136–141, 1973.

[Eswaran, *et al.* 1976] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976. Also published as IBM RJ1487, December, 1974.

[Fredman & Tarjan 1984] Michael L. Fredman and Robert Endre Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. January 1984. Version of the paper that was used in MIT course 6.851.

[Gifford & Donahue 1985] D. K. Gifford and J. E. Donahue. Coordinating Independent Atomic Actions. In *Proc. of IEEE CompCon85*, pages 92–95, IEEE, February 1985.

[Gifford 1979] D. K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–162, ACM SIGOPS, Pacific Grove, CA, December 1979.

[Gray 1978] Jim Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems: An Advanced Course*, chapter 3.F, pages 394–481, Springer-Verlag, 1978. Also appears as *IBM Research Report RJ 2188*, Aug., 1987.

[Gray, *et al.* 1976] J. N. Gray, R. A. Lori, G. F. Putzolu, and I. L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In G.M. Nijssen, editor, *Modeling in Data Base Management Systems*, pages 365–394, North Holland, Amsterdam, 1976. Also available as IBM Research Report RJ 1654/1706.

[Gruber 1989] Robert Edward Gruber. *Optimistic Concurrency Control For Nested Distributed Transactions.* Master's thesis, MIT, June 1989.

[Herlihy, *et al.* 1987] M. Herlihy, N. Lynch, M. Merritt, and W. Weihl. On the correctness of orphan elimination algorithms. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, IEEE, Pittsburgh, July 1987.

[Kolodner 89] Elliot K. Kolodner. *Design Bug in Argus Recovery System.* DSG Note 158, Programming Methodology group, L.C.S, M.I.T, November 89.

[Lampson & Sturgis 1976] Butler W. Lampson and Howard E. Sturgis. Crash Recovery in a Distributed Data Storage System. 1976. version of paper that was not published.

[Liskov & Guttag 1986] B. H. Liskov and J. V. Guttag. *Abstraction and specification in program development.* MIT Press, 1986.

[Liskov & Scheifler 1983] Barbara Liskov and Robert Scheifler. Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[Liskov & Shrira 1988] B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proc. of the ACM SIGPLAN '88 Conference on Programming Languages Design and Implementation*, ACM, June 1988.

[Liskov 1984] Barbara Liskov. *Overview of the Argus Language and System.* Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, February 1984.

[Liskov, *et al.* 1987a] B. Liskov, R. Scheifler, E. F. Walker, and W. Weihl. Orphan Detection (Extended Abstract). In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, IEEE, July 1987. Also appears as Programming Mothodology Group memo 53, M.I.T, L.C.S, Feb. 87.

[Liskov, *et al.* 1987b] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proc. of the 11th Symposium on Operating Systems Principles*, ACM, Austin, Tx, November 1987.

[Menasce & Muntz 1979] D. Menasce and R. Muntz. Locking and deadlock detection in distributed data bases. *IEEE Transactions on Software Engineering*, SE-5(3):195–202, May 1979.

[Moss 1981] J. Elliot B. Moss. *Nested transactions: an approach to reliable distributed computing.* Ph.D. thesis, Massachusetts Institute of Technology, 1981. Available as Technical Report MIT/LCS/TR-260.

[Nguyen 1988] Thu Duc Nguyen. *Performance Measurements of Orphan Detection in the Argus System.* Master's thesis, M.I.T, June 1988.

[Obermarck 1982] R. Obermarck. Distributed deadlock detection algorithm. *ACM Transactions on Database Systems*, 7(2):187–208, June 1982.

[Oki, Liskov & Scheifler 1985] Brian M. Oki, Barbara H. Liskov, and Robert W. Scheifler. Reliable Object Storage to Support Atomic Actions. *ACM SIGOPS Operating Systems Review*, 19(5):147–159, December 1985. Proceedings of the Tenth ACM Symposium on Operating Systems Principles, 1-4 December 1985, Orcas Island, Washington, U.S.A. Also appears as Programming Methodology Group memo 45, M.I.T, L.C.S, Sept. 85.

[Oppen & Dalal 1981] D. C. Oppen and Y. K. Dalal. *The Clearinghouse: a decentralized agent for locating named objects in a distributed environment.* Technical Report OPD-T8103, Xerox Office Products Division, October 1981.

[Perl 1987] Sharon E. Perl. *Distributed commit protocols for nested actions.* Master's thesis, MIT, October 1987. Available as MIT/LCS/TR-431, November 1988.

[Postel 1980] J. Postel. *Internet User Datagram Protocol.* Request For Comments 768, University of Southern California / Information Sciences Institute, August 1980.

[Reed 1978] D.P. Reed. *Naming and synchronization in a decentralized computer system.* Ph.D. thesis, Massachusetts Institute of Technology, 1978. Available as Technical Report MIT/LCS/TR-205.

[Reed 1983] David P. Reed. Implementing Atomic Actions on Decentralized Data. *ACM Transactions on Computer Systems*, 1(1):3–23, Februar 1983.

[Schlichting & Schneider 1983] R. D. Schlichting and F. B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computing Systems*, 1(3):222–238, August 1983.

[Schwarz 1984] Peter M. Schwarz. *Transactions on typed objects.* Ph.D. thesis, CMU, December 1984. Available as Technical Report CMU-CS-84-166.

[Skeen 1981] D. Skeen. Nonblocking commit protocols. In *Proceedings of a Symposium on the Management of Data*, pages 133–142, ACM SIGMOD, 1981.

[Spector, *et al.* 1987] Alfred Z. Spector, Dean Thompson, Randy F. Pausch, Jeffery L. Eppinger, Dan Duchamp, Richard Draves, Dean S. Daniels, and Joshua J. Bloch. *Camelot: A Distributed Transaction Facility for Mach and the Internet–An Interim Report.* Technical Report CMU-CS-87-129, CMU, June 1987.

[Thomas 1979] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[Walker 1984] Edward Franklin Walker. *Orphan Detection in the Argus System.* Technical Report 326, MIT/LCS, June 1984.

[Weihl 1984] William E. Weihl. *Specification and implementation of atomic data types.* Ph.D. thesis, MIT, March 1984. Available as MIT/LCS/TR-314.

[Xu & Liskov 1988] Andrew Xu and Barbara Liskov. *Stream Calls and Locking.* DSG note 150, MIT, January 1988.

[Zhao & Ramamritham 1985] Wei Zhao and Krithivasan Ramamritham. *Use of Transaction Structure for Improving Concurrency.* COINS Technical Report 86-2, University of Massachusetts, March 1985.