LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TR-453

OPTIMISTIC CONCURRENCY CONTROL FOR NESTED DISTRIBUTED TRANSACTIONS

Robert Edward Gruber

June 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

Optimistic Concurrency Control for Nested Distributed Transactions

by

Robert Edward Gruber

June 1989

© Massachusetts Institute of Technology 1989

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract 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

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

Optimistic Concurrency Control for Nested Distributed Transactions

by

Robert Edward Gruber

Abstract

Optimistic concurrency control techniques allow atomic transactions (or *actions* for short) to execute without synchronization, relying on commit-time validation to ensure serializability. Previous work in this area has focussed on single-level actions. This thesis extends previous work on optimistic concurrency control to distributed systems with *nested* actions.

The thesis presents two contrasting models for managing nested actions, which we call the *fixed action model* and the *fixed object model.* In the fixed action model, an action executes at only a single network node; if an action accesses an object whose storage is provided by another node, it brings a copy of the object to its node. In this model, caching copies of non-local objects can be used to improve performance by reducing the number of non-local object requests. We show that optimistic concurrency control has an advantage over pessimistic concurrency control with respect to this object caching. In the fixed object model, actions can span network nodes: an action at one node can start a nested action at another node. However, an object's state is never moved from the node providing its storage. While there is a clear reason for using optimistic concurrency control for systems following the fixed action model, there is no clear reason for choosing either optimism or pessimism for systems following the fixed object model; however, the development of this model is an important step in the study of hybrid models.

Keywords: Concurrency control, Optimistic concurrency control, Transactions, Nested transactions, Atomicity, Commit protocols, Distributed computer systems.

This report is a minor revision of a Master's thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on May 12, 1989, in partial fulfillment of the requirements for the degrees of Master of Science in Computer Science and Engineering and Bachelor of Science in Electrical Engineering and Computer science. The thesis was supervised by Professor Barbara Liskov. Part of the research was done at the Xerox Palo Alto Research Center under the supervision of Frank Zdybel.

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

Acknowledgments

There are many people who helped make this thesis a reality and who contributed to my enjoyment of the research. In particular, I would like to thank the following people:

Barbara Liskov, for her welcome, support, and counsel; and for her efforts with respect to the thesis.

Frank Zdybel, for his encouragement and friendship; and for plowing through the inevitable red tape.

Dave Robson, John Tucker, and Joel Moses, for their friendship; and for their willingness to listen and lend a helping hand.

Wilson Hsieh, for reading of a draft of the thesis and helping with the figures. His many helpful comments and questions improved the presentation.

Carl Waldspurger, for helping me put the final touches on the thesis at six in the morning.

Jeff Cohen, for his wise-cracks and stoned wheat crackers.

All of the members of the System Concepts Lab at PARC and the Programming Methodology Group at MIT, for providing two fun research environments and many interesting discussions.

Past and present members of Conner Three, Corby's Kids, and the people who hang out in the Campus Room (you know who you are), for a fun six years.

Finally, I'd like to thank my family for their love and support.

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

Contents

List of Figures

Chapter 1

Introduction

Optimistic concurrency control techniques allow atomic transactions (or *actions* for short) to execute without synchronization, relying on commit-time validation to ensure serializability. Previous work in this area has focused on single-level actions for both centralized and distributed systems. This thesis extends previous work on optimistic concurrency control to distributed systems with *nested* actions.

The thesis presents two contrasting models for managing nested actions, which we call the *fixed object model* and the *fixed action model.* These models mirror the two basic approaches that have been taken by the designers of transaction systems. In the fixed action model, an action is restricted to execute at a single network node; if an action accesses an object whose storage is provided by another node, the action brings the object's state to its node. When an action commits, its updates are sent to the nodes where the updated objects are located so that those nodes can install the updates. In the fixed object model, actions can span network nodes: an action at one node can move to another node. However, an object's state is never moved from the node providing its storage; all accesses occur locally, and at commit time each node where updates were performed is asked to install those updates. Informally, in the fixed action model the objects move to the actions, while in the fixed object model the actions move to the objects.

While there have been numerous studies comparing optimistic techniques to the more common *pessimistic* techniques, these studies have focussed on single-site systems, or on systems similar to the fixed object model. As a result, they have ignored a significant advantage of optimistic concurrency control for systems similar to the fixed action model. In that model, it is natural for nodes running actions to cache copies of non-local objects, so that multiple accesses to a given object (by one or more actions) will require only a single network request for the object's state. In a pessimistic system, it would be necessary to keep such object caches coherent. In practice, this would mean requesting a non-local lock before allowing an access to a local cached object. Since a lock request is almost as costly as requesting the entire object, caching in a pessimistic system will not significantly improve performance. In an optimistic system, however, there is no coherency requirement: an out-of-date object can be read, since commit-time validation will detect this. Thus, cached values can be accessed asynchronously with respect to the commit of object updates, and caching should result in significant performance gains.

For systems like the fixed action model, optimistic concurrency control has an advantage over pessimistic concurrency control. For systems following the fixed object model, although there are currently no clear reasons for chosing one concurrency control method over the other, the development of this model is an important step in studying two interesting hybrid models — a combination of the fixed action and the fixed objects models, and a hybrid pessimistic-optimistic system with fixed object restrictions.

The next section provides motivation for nested actions and describes in more detail why it is advantageous to use optimistic methods for systems like the fixed action model.

1.1 Motivation

Atomic transactions (or *actions* for short) are a mechanism for building reliable distributed systems in the presence of failures. They provide two important properties: *recoverability* and *serializability*. Recoverability means that actions have an "all-or-nothing" behavior: an action either executes to completion, in which case we say it *commits*, or it has no effect on the persistent state of the system, in which case we say it *aborts*. Serializability means that the actual effect of executing actions concurrently is equivalent to the effect of executing those actions in some serial order.

Recoverability protects us from failures, while serializability allows us to reason about concurrency by considering the effect of each action separately. For example, consider a banking system, where the persistent state of the system is a set of account balances. To transfer money from account A to account B, we test balance A to see if there is enough money: if there is, we decrement A and increment B. If a failure were to occur midway through this process, money could be lost from the system. However, if we run the test, decrement and increment as a single action, then the recoverability property guarantees that either the action will commit, in which case both accounts are updated, or the action will abort, in which case no change to the accounts occurs. If two identical transfers from

A to B run concurrently, we might worry that both could pass the test for sufficient funds (observing the same initial value), even if there were not enough money for both transfers. However, by running each transfer as an action, serializability guarantees that the effect of these actions must be equivalent to running one action and then the other, in some order, so we do not have to concern ourselves with potential problems arising from concurrent execution.

Nested actions (or *subactions*) are a natural generalization of top-level actions (or *topactions*) as described above. A subaction is an action that is started within another action. Nesting can occur to an arbitrary depth; the resulting structure can be described using *action trees*, and we use standard tree terminology (such as parent, child, sibling, and so on) when discussing relationships between actions. Like the topactions described above, subactions have the recoverability and serializability properties. Usually, an action that starts some subactions will wait for them to complete (commit or abort) before continuing. A subaction fails independently of its parent action, providing a kind of checkpointing mechanism: the work done by a parent action prior to starting a child will not be lost if the child aborts. A parent action can start one subaction at a time, forcing the subactions that it executes to run in a given order, or it can start several subactions at the same time, allowing them to run concurrently. The children of a given parent action are serialized with respect to each other, *i.e.*, each set of *sibling actions* is serialized. Thus, just as topactions allow us to reason about concurrency at the system level, subactions allow us to reason about concurrency within an action.

The mechanism in an action system that guarantees the serializability property of actions is called a *concurrency control mechanism*. Two basic approaches to concurrency control have been studied: *pessimistic concurrency control* and *optimistic concurrency control*. Herlihy [Herlihy 1986] has informally described the difference between the two approaches as the difference between *asking permission first* and *apologizing later*. In a pessimistic system, we assume that actions will conflict, and we use some mechanism (such as a locking mechanism) to ensure that the actions in the system can always be serialized — as an action executes, it is not allowed (*e.g.*, because of the locking rules) to conflict with any other active action, and all actions can therefore be serialized. In an optimistic system, we assume that actions will not conflict; concurrent actions are allowed to execute without any synchronization. When an action attempts to commit, it enters a *validation phase* where we verify our optimistic assumption. An action passes validation only if it can be serialized with respect to other actions in the system. Actions that pass validation are allowed to commit; the others are aborted and restarted.

In pessimistic systems, actions can delay unnecessarily. In addition, pessimistic systems can get into deadlock situations, where a group of actions is unable to proceed because each action in the group is waiting for a lock held by some other action in the group. In optimistic systems, actions that fail validation abort and restart, redoing work that would not be redone in a pessimistic system. In addition, while optimistic systems can not deadlock, they can suffer from *livelock*: a long-lived action can continually abort and restart, failing validation each time because of a continuous series of short-lived actions that conflict with it. A number of studies [Agrawal 1983, Badal 1981, Carey 1983, Franaszek & Robinson 1985, Menasce & Nakanishi 1982, Tay *et al.* 1984] have attempted to compare the performance of the two approaches; however the results are still inconclusive. Both approaches work well when there are few conflicting actions in the system, while neither approach works very well when there are many conflicting actions in the system. At best, we can say that pessimistic systems seem to be slightly more robust under a wider range of system parameters (average number of concurrent actions, percentage of read-only actions, average length of an action, *etc.*).

The above studies have not considered systems like the fixed action model, where we do *object caching*. In these systems, there is a distinct advantage to using an optimistic approach. In the fixed action model, when an action accesses a non-local object, it acquires a copy of the object by sending a request over the network. Thus, an action brings to one network node copies of all of the objects that it accesses. Since a number of different actions running at the same node might access an object within a short span of time, it makes sense for a node to cache the object copies that it has requested. If a non-local object's state is cached locally, an action would like to access this state rather then using a network request. However, the choice of concurrency control mechanism affects the performance of accessing a cached object. In a pessimistic system with locking, it is necessary to acquire the appropriate lock (by sending a lock request) before accessing an object. Thus, a round-trip network delay is required even when reading a cached object. This delay is necessary to ensure two things: the action must read (or modify) an up-to-date copy of the object, and the locking rules must be maintained. Note that a network lock request is almost as costly as a request for a copy of the object, since the cost of a message is independent of message size, for objects of reasonable size. Thus, object caching becomes almost useless in a pessimistic system. In contrast, in an optimistic system an action can read an out-of-date object copy; this is exactly what validation is designed to detect. Thus, an action can read cached objects without using any network messages.

In fixed action systems, optimism helps us with two factors that play important roles in distributed system performance. One factor is delay due to synchronization with a remote location: where possible, we would like to avoid having to block while a round-trip message is sent to another node. Above, we saw that optimism allows us to avoid the delay associated with sending lock requests. The second factor is the number of messages that have to be processed by the system: as more messages are sent, message transmission times and message processing times both increase. In a pessimistic fixed action system, we would have to send one message for each lock request, along with some messages at action commit time. With an optimistic approach, we would only send messages at commit time, assuming all objects needed by the action were already cached. (The optimistic commit-time messages would be larger than the pessimistic commit-time messages, since they must include some extra information used for validation. However, the number of messages is more important than the size of each message.)

An interesting hybrid optimistic-pessimistic method is used by Symbolic's Statice system [Weinreb *et al.* 1988]. Statice is essentially a pessimistic version of the fixed action model. As an action reads or writes pages, the pages are sent to the action's node; nodes cache pages across actions to reduce the number of network page requests. The system uses read-write locking, so it is mostly a pessimistic system. However, when accessing a cached page, it optimistically assumes that it will be able to get the lock for the page — it sends a lock request, but does not wait for the response before reading the page. If the lock request is denied, the requesting action is aborted and restarted [Gerson 1989]. Thus, Statice is using optimism in one of the two places where it has an advantage over pessimism: it avoids the synchronization cost of lock requests. Using a fully optimistic approach results in fewer messages, since the lock requests do not need to be sent. (However, it should be possible to "piggy-back" lock requests on other messages, such as commit-time messages or requests for pages that are not cached. Alternatively, several lock requests could be grouped into a single message. Using optimism, the lock requests become asynchronous, and this kind of piggy-backing is possible.)

There is currently no fully optimistic system that supports subactions. In fact, the combination of optimism and nested actions has never been described in the literature.

In this thesis, we explore this combination. As we have shown here, the most interesting application of optimism will be in systems like the fixed action model. A number of groups are currently (or will soon be) building object repositories, or object-oriented databases (for example, see [Hornick & Zdonik 1987] and [Moss & Sinofsky 1988]). These object repositories are similar to the fixed action model; the designers of these repositories will clearly be interested in that model. Caching will be used, and, as we argue here, an optimistic or hybrid approach is called for.

For models such as the fixed object model, there is currently no clear reason to consider an optimistic approach over a pessimistic one. The model is included in the thesis for two reasons. First, developing the model allows us to compare it to current pessimistic approaches with the same fixed object restrictions. We believe that there are no clear *disadvantages* to using optimism in a system with fixed object restrictions. Second, although it may not be worthwhile to use strict optimism for a pure fixed object model, we would eventually like to explore two interesting combinations: the combination of the fixed action and fixed object models, and the combined use of optimism and pessimism in a system with fixed object restrictions. By developing the fixed object model, we have a starting point for exploring these hybrid models.

1.2 Related Work

The earliest concurrency control scheme to use validation may be that of [Thomas 1978]. Kung and Robinson proposed a centralized optimistic concurrency control method and gave arguments for such an approach in [Kung & Robinson 1981]. Since then a number of papers have been written that have: extended optimistic concurrency control to distributed systems [Ceri & Owicki 1982]; proposed new validation techniques that can serialize more actions than Kung and Robinson's method [Lausen 1983, Herlihy 1986]; studied alternative validation methods [Harder 1984]; and compared optimistic and pessimistic techniques [Agrawal 1983, Badal 1981, Carey 1983, Franaszek & Robinson 1985, Menasce & Nakanishi 1982, Tay *et al.* 1984].

Nested actions are proposed in [Davies 1978, Reed 1978]. Several approaches to using pessimistic concurrency control for nested actions have been studied. Read-write locking for nested actions is described in [Moss 1981] and has been implemented by systems such as Argus [Liskov 1984, Liskov *et al.* 1987a] and Camelot [Spector *et al.* 1987, Spector & Swedlow 1987], using locking approaches that are variants of Moss's approach.

Reed describes a static timestamping approach to concurrency control for nested actions in [Reed 1983].[1] Recently, a formal model for studying nested actions has been developed [Lynch *et al.* 1988].

Lausen has proposed a concurrency control method that combines optimistic and pessimistic techniques [Lausen 1982]; recent object repository (object-oriented database) systems, such as ObServer [Hornick & Zdonik 1987] and Mneme [Moss & Sinofsky 1988] provide support for this sort of combination.

As we mentioned above, Symbolic's Statice system [Weinreb *et al.* 1988] uses a hybrid optimistic-pessimistic approach, where reading cached pages occurs asynchronously with respect to page updates [Gerson 1989]. Servio Logic's Gemstone system [Maier *et al.* 1986] is a fully optimistic system that is similar to the fixed action model and does caching as we described above. It is the only fully optimistic system we know of that follows the fixed action model; however, there are two differences between our model and the Gemstone system: Gemstone does not have nested actions, and there is only a single network node where persistent objects are stored. The fixed action model allows for any number of nodes to store objects.

To improve the performance of read-only actions, and to ensure that actions always see a consistent state of the objects in the system, it is possible to retain multiple versions of committed objects. Multi-version schemes have been studied for both optimistic [Agrawal *et al.* 1987] and pessimistic [Weihl 1986] systems. The Agrawal paper also discusses parallel validation issues; all of our parallel validation algorithms are based on ideas presented there.

An important part of a concurrency control mechanism is its *commit protocol*, which in a distributed system is a protocol that ensures that the nodes participating in an action either all commit or all abort. To commit top-level actions, our models use a variation of a standard two-phase commit protocol [Gray 1979, Mohan & Lindsay 1983, Lampson 1981, Lindsay *et al.* 1979, Lindsay *et al.* 1984]. Non-blocking and three-phase commit protocols have also been studied [Skeen 1981, Dwork & Skeen 1983].

Two-phase commit and the other protocols mentioned above ensure that all partici-

[1] In a timestamping approach, an action's serialization order is chosen when it is created; at each read or write, the action verifies that its intended access is consistent with this order. This scheme is similar to both pessimistic and optimistic schemes. Like pessimistic schemes, an action is synchronized at each step; an action can end up delaying, waiting for another action to commit or abort. Like optimistic schemes, there can be no deadlocks, but there can be livelock, where an action is continually aborted and restarted.

pants find out about the agreed-upon outcome. Such a protocol is clearly necessary for topaction commit. However, these protocols are expensive, and we would like to avoid using them for subaction commit. It is not necessary for all participants in a subaction to learn about its outcome at the time that the subaction commits or aborts. All participants will eventually learn the outcome at topaction commit time; thus the subaction commit protocol need not guarantee the same thing, and we can design protocols for subaction commit where participants of a subaction learn about its outcome in a "lazy" fashion. Special protocols for subaction commit have been studied for fixed object pessimistic systems [Liskov *et al.* 1987a, Perl 1988].

1.3 Roadmap

The remainder of the thesis is organized as follows. Before presenting our models, we describe in Chapter 2 the model of nested action systems upon which we base our protocol. We then use a process of step-wise refinement to present our models. Chapter 3 presents a basic optimistic concurrency control model for a single-site system with topactions only. It describes optimistic concurrency control in more detail and introduces relevant terms. Chapter 4 extends this model to include nested actions, still for a single-site system. We then present our distributed system models: Chapter 5 presents the fixed action model, while Chapter 6 presents the fixed object model. Finally, Chapter 7 gives a brief comparison of the two distributed models, summarizes the thesis, and mentions some areas for future work.

Chapter 2

Definitions and Model

In this chapter we define actions, nested actions, and related terms, and we describe a high-level model of a distributed system, which we use for our distributed models.

2.1 Low-level System

At a lower level of abstraction, a distributed system is a collection of nodes connected by a communications network. Distinct nodes communicate with each other by sending messages (of unrestricted length) over the network. Nodes are typically individual computers. Nodes may crash or otherwise fail, although we do assume that when a node fails any messages it sends are detectably invalid. The network may fail by partitioning; messages may be lost, duplicated, delayed, or delivered out of order. We assume that failures are eventually repaired: nodes eventually recover from crashes, and partitions are eventually mended. Nodes have access to both volatile and stable storage. Volatile storage is lost in a crash, while stable storage is intact upon the recovery of a node [Lampson 1981].

2.2 High-level System

At a higher level of abstraction, we view a distributed system as a collection of active communicating *entities*, where each entity resides at a single node in the network. Entities are *resilient*; with high probability, they survive crashes of their nodes (this is done, *e.g.*, by storing in stable storage information necessary for recovery).

Each entity has a message interface, which describes a set of messages to which it will respond. Each message has a set of inputs and results, and is similar to an operation of an abstract data type. Message invocation follows the semantics of *remote procedure calls*, with call-by-value semantics. Unless we state otherwise, message invocation is assumed to use a communication mechanism that reliably delivers a call and corresponding response. (Reliable delivery has a cost associated with it; where appropriate, we point out cases where it is not required.)

Each entity E has a globally unique identifier id(E).

2.3 Objects and Object Managers

Objects encapsulate data. For simplicity, our models assume that there are two types of operations on objects, read operations and write operations. A read operation observes the state of an object, but does not change it, while a write operation modifies the state of an object. Our optimistic concurrency control algorithms are based on the reads and writes that an action performs; *i.e.*, they use read-write conflict detection.

One type of entity that will exist in some form in all of our models is an *object manager* entity, or OM. Although OM's will differ from model to model, all OM's have some things in common. Each OM manages a set of persistent objects, storing the state of each object in stable storage. (Normally, an OM's stable storage will be provided by the node where the OM is running. There are other possibilities; for example, stable storage might be provided as a network service [Daniels *et al.* 1987, Cohen 1989].)

An object *O* managed by object manager *OM-X* has an identifier, *id(O)*, that is unique across all object managers. (A typical approach is to form *id(O)* from *id(OM-X)* and an additional identifier that is unique within *OM-X*.) Associated with each object identifier is a version of the object that is stored in stable storage. This version is called the *base version* of the object. An OM can atomically update a set of its objects: if an update succeeds, each object's base version is replaced with a new base version; if the update fails, all of the old base versions are unchanged.

2.4 Actions and Distributed Action Systems

Atomic transactions are a mechanism for maintaining the consistency of data in the presence of concurrency and failures. While actions are important in centralized systems, and have been used in database work for quite some time, they are particularly useful in distributed systems where the failure modes are more numerous and complex. Programs that execute at multiple entities may partially fail, if only some of the entities crash. In addition, communication failures (such as network partitions) may prevent such programs from running to completion. Actions provide a means for the programmer to cope with such failures.

A distributed action system is composed of a set of communicating entities that support actions. Some or all of these entities will be object managers. An action runs as a process within an entity, and delimits a computation over a set of objects managed by one or more OM's. Two properties of actions, *recoverability* and *serializability*, ensure that the state of the objects in the system remains consistent in the presence of failures. Recoverability means that an action either runs successfully to completion (it *commits*) or else has no effect at all (it *aborts*). If a failure occurs during an action, the action may simply abort, undoing all modifications that the computation has made. Serializability means that when actions are executed concurrently, the effect is as if they were run sequentially in some order. This allows a programmer to ignore concurrency when reasoning about the effects of an action.

Transaction systems usually provide a third property for actions, *persistence*. Persistence means that the system ensures with very high probability that the effects of actions that commit will not be lost due to failures. Our models provide persistence by storing object state (base versions) and commit information in stable storage.

Our models only guarantee serializability for committed actions. An action that ends up aborting has no effect on the persistent state of objects; however, while it is executing it may observe an object state that is not consistent with a serial execution of the actions in the system. (This is only a problem for actions that perform operations that cannot be undone if the action aborts, such as writing to an output device.)

2.4.1 Nested Actions

A *nested* action is an action that is started from within another action. Actions can be nested to arbitrary levels, forming action trees with *topactions* at the roots. Actions that are not topactions are called *nested actions* or *subactions*. We use standard tree terminology to describe the relationships between actions, such as *parent, child, sibling, ancestor* and *descendant*. For convenience, we define an action to be its own ancestor and descendant, while an action's *proper ancestors* and *proper descendants* do not include the action itself.

A nested action system will have one topaction tree for each topaction in the system. It can be useful to consider the set of all topactions in the system to be siblings with a common parent; the common parent is the root of a single *global action tree*.

Actions are named by *action identifiers*. An action identifier is structured and contains information about the action tree of the action it describes. In particular, an action identifier for an action contains a name for the action, the entity at which the action executes, and

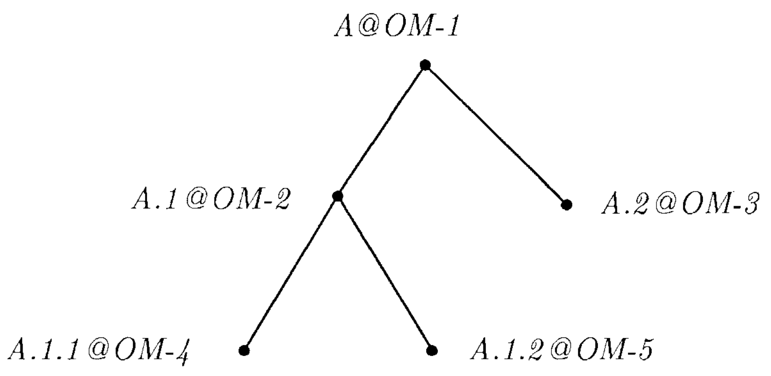


Figure 2.1: An Action Tree.

(except for topactions) the identifier of the action's parent action. (Thus, an action identifier includes the action identifiers of all of its proper ancestors.) For example, in Figure 2.1 we show an action tree for topaction A: each action identifier indicates at what entity the action is running, and each subaction's identifier should be thought of as including its parent's identifier, which we indicate by using a nested numbering scheme. (In the figure, each action is running at a different entity. This can only occur in the fixed object model only; in the fixed action model, all actions in a topaction's tree run at the same entity.)

Nested actions are useful for obtaining checkpoints and managing concurrency within an action. A subaction fails (aborts) independently of its parent. Thus, an action can create a subaction to attempt some computation, and if that computation fails, either go on with other computations or create another subaction to retry the computation. In this way, the parent's state at the time it starts the subaction is "checkpointed".

Concurrency within an action is obtained by allowing a parent to start concurrent subactions. While a child action is running, its parent is suspended. A parent action can choose whether or not to run its children concurrently. For example, if action *A* in Figure 2.1 wanted *A.1* to execute strictly before *A.2*, it would first start just *A.1*: *A* would suspend until *A.1* (and its descendants) had committed or aborted up to *A*. After *A* resumed, it would start *A.2*. If *A* wanted to execute both *A.1* and *A.2* concurrently, it could start them both; it would be suspended until they both either committed or aborted.

Siblings are serialized at each level of the action tree. Thus, concurrent siblings do not interfere with one another.

In the fixed action model, all descendants of a topaction must execute at the same entity as the topaction; a parent action always knows when its child has finished executing. In the fixed object model, however, a parent action at one entity can start a subaction at another entity; it then waits for a message from the other entity indicating that the subaction is finished. This message may never arrive, because of network or node failures; the parent can decide (after waiting for some time period) to abort the subaction and continue.

The commit of a subaction is always relative to its parent. If a subaction commits and its parent aborts, the effects of the subaction will be undone. When an action A and all its ancestors up to its topaction commit, we say that A has committed *to the top.* If A's topaction then commits, we say that A has committed *through the top.*

Chapter 3

Basic Single-Site Model

In this chapter, we describe a basic optimistic concurrency control model that uses read-write conflict detection. The model is for single-level actions (topactions) executing in a single-site transaction system. It is based on the original optimistic algorithm proposed in [Kung & Robinson 1981]. In that algorithm, an action's start time is used to represent the read time of each object in the action's read set. In fact, for a long-lived action, the read time for a given object may be significantly later than the action's start time. For this reason, our basic model keeps track of an initial read time for each object in an action's read set. While this takes more space, it prevents some actions from being aborted unnecessarily. Except for this change to the algorithm, the model presented here is simply a re-expression of prior work; our original work begins in the next chapter. Kung and Robinson present both a serial and a parallel validation algorithm; we do the same here. All of our models owe a great deal to a paper by Agrawal *et. al.* [Agrawal *et al.* 1987], which presents optimistic algorithms for a single-level multiversion system. In particular, we use the same approach to parallel validation.

3.1 Shadow Versions

In this and the following chapter, we present models for a centralized system. A centralized system consists of one entity, an OM executing at a single node. The OM manages both stable storage and all actions in the system. Each object O managed by the OM has a unique identifier, id(O). The state of O that is stored in stable storage is called the *base version* of O.

In the model presented in this chapter, only topactions are allowed. A topaction begins, reads and writes one or more objects, and then attempts to commit. (A topaction can also explicitly abort.) If a topaction does not perform any write operations, we say that it is read-only; otherwise we say that it is read-write.

The updates of a read-write topaction are not performed directly on the base versions of the objects that it modifies. Instead, at the time when a topaction T first attempts to write an object O, a private copy of the base version of O is created for T's use, and T makes all modifications on this private copy. We call this copy a *shadow version;* while T is executing, T's version of O *shadows* the base version, meaning that T reads and writes the shadow version rather than the base version. If T aborts, all of its shadow versions are thrown away, and the persistent state of the database (the state of the base versions) is not changed. If T commits, its shadow versions replace the base versions that they are shadowing. Installing shadow versions is done atomically: either it succeeds completely, or not at all. Thus, shadow versions provide the *recoverability* property of actions. If an action commits, all of its updates are applied; otherwise none of them are.

Note that an action can only read the base versions of objects and its own shadow versions. Thus, if a topaction *T1* observes a modification made by another topaction *T2* to an object O, *T2* must have committed before *T1* read O.

Figure 3.1 gives an example of the use of shadow versions. Part (a) shows a system with three objects, x, y, and z, and two topactions, A and B. The current base versions for x, y, and z are integer values (objects can be of any type in general). Topaction A has modified objects x and y (it has incremented each of them), while topaction B has modified object z. At this point, neither action has attempted to commit. Ignoring the reasons, suppose action A commits and action B aborts. Part (b) shows the system after action A commits: A's shadow versions of x and y have been moved up to become new base versions. Part (c) shows the system after action B aborts: its shadow versions have been thrown away, and the base version for object z is unchanged.

3.2 Overview of Validation

When a topaction T attempts to commit, it enters its *validation phase.* If it passes validation, it is allowed to commit; otherwise it is aborted, and its shadow versions are discarded. An action that fails validation will usually be restarted; *i.e.*, a new action will be started that attempts to perform the same computation. If a read-write action commits, it enters an *update phase*, where it installs its shadow versions as base versions.

Validation ensures that the net effect of all of the topactions that are allowed to commit is the same as the net effect of executing those topactions in some non-overlapping

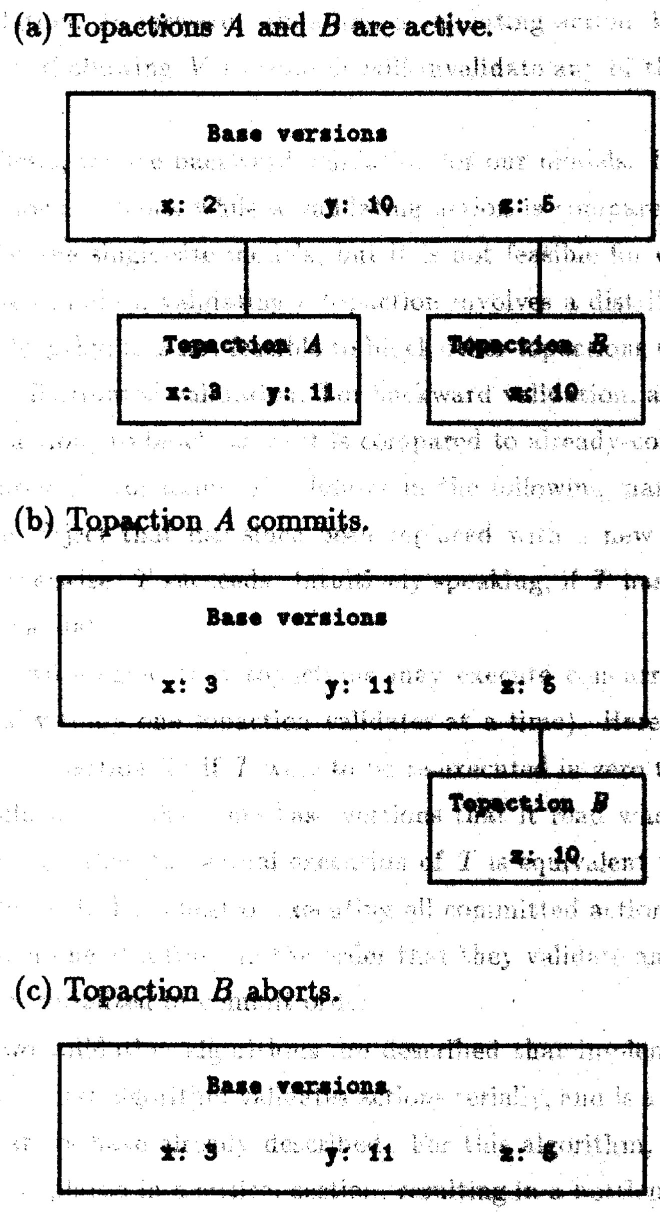


Figure 3.1: Shadow Versions — Basic Single-Site Model

order. There are two basic approaches to validation: *backward validation* and *forward validation* [Harder 1984]. In backward validation, a validating action V is compared to already-committed actions; if any of these already-committed actions has invalidated V, V fails validation. In forward validation, a validating action V is compared to currently active actions; if allowing V to commit will invalidate any of these active actions, V fails validation.

In this thesis, we use backward validation for our models. Forward validation requires all active actions to block while a validating action is compared to them. This might be reasonable for the single-site models, but it is not feasible for our distributed models. In our distributed models, validating a topaction involves a distributed commit process that can be quite lengthy; it is not feasible to block other topactions whenever some topaction is performing a distributed validation. For backward validation, a validating action does not cause active actions to block, since it is compared to already-committed actions.

In this model, a topaction T validates in the following manner: if T has read a base version of an object that has since been replaced with a new base version, then T fails validation; otherwise, T succeeds. Intuitively speaking, if T has read an out-of-date value, then T fails validation.

For now, we assume that topactions may execute concurrently, but that they must validate serially (only one topaction validates at a time). Here is another way of viewing validation for topaction T: if T were to be re-executed in zero time, at the current instant in time, would it read the same base versions that it read when it actually executed? If the answer is yes, then the actual execution of T is equivalent to executing T at the time that it is validated. The effect of executing all committed actions is therefore equivalent to executing them one at a time, in the order that they validate and commit: we say that the topactions are serialized in commit order.

Below, two validation algorithms are described that implement this simple validation technique. The first algorithm validates actions serially, and is a straightforward implementation of what we have already described. For this algorithm, the validation and update phases must be placed in a critical section, resulting in a bottleneck. The second algorithm validates actions in parallel, thus avoiding the bottleneck.

```
Topaction T reads Object O:
    if id(O) $\notin$ rs(T) then add the pair [id(O), tnc] to rs(T) end
    if T has a shadow version of O then
            read the shadow version
        else
            read the base version
        end

Topaction T writes Object O:
    if id(O) $\notin$ ws(T) then
        add id(O) to ws(T)
        create a new shadow version of O for topaction T from O's base version
        end
    modify T's shadow version of O
```

Figure 3.2: Read and Write — Basic Single-Site Model

3.3 Data Structures

The following data structures need to be maintained. First, there is a monotonically increasing transaction counter, *tnc*, which keeps a count of committed read-write actions. For each topaction T, there is a read set *rs(T)* and a write set *ws(T)*. Finally, for validation purposes, the set *rw-committed* contains information about each committed read-topaction: for read-write action RW, it contains a pair of the form `[ws(RW), et(RW)]`, where *ws(RW)* is RW's write set and *et(RW)* is RW's end time. (A read-write action's end time is chosen when its updates are installed.)

3.4 Read and Write

Figure 3.2 summarizes the rules for reading and writing an object. When topaction T first reads an object O, the pair `[id(O), tnc]` is added to T's read set. This pair indicates what object was read and when; the transaction counter is used as a "clock" for recording read times. Our algorithm differs from Kung and Robinson's: their read sets only contain object identifiers, and the action's start time is used as the read time of each object in *rs(T)*. Their algorithm works because the start time is an underestimate of the actual read time; however, actions can unnecessarily fail validation because of this simplification. When T first writes an object O (and a shadow version is created), *id(O)* is added to T's write set (no write time is recorded).

To make our validation algorithms simpler, we assume throughout the thesis that an object is always read before it is written: actions do not perform "blind writes". This is not a limiting assumption; to include blind writes, we could simply add an appropriate entry to an action's read set before performing a blind write. (We did not take this approach because would complicate the descriptions of the write operation.)

3.5 Serial Validation

Pseudo-code for our serial validation algorithm is given in Figure 3.3. The algorithm works as follows. When a topaction T attempts to commit, it enters validation. T's start time, *st(T)*, is set to the earliest read time in *rs(T)*. T is then compared to each committed read-write action RW that ended after T started. If an object in T's read set is also in RW's write set, and if RW committed (installed its write to the object) after the read occurred, then T fails validation: it will abort and restart. If T does not fail validation, it commits. When a read-write action T commits, it is assigned an end time, *et(T)* (*tnc* is incremented, and T's end time is set to *tnc*'s new value), its shadow versions are installed as base versions, and a pair containing its end time and write set is added to the set *rw-committed*. (When a read-only action passes validation, it has no further work to do to commit.)

This algorithm uses a single critical section to ensure that two topactions cannot execute the validation and update process concurrently. This creates a bottleneck, and a parallel algorithm is clearly needed.

Figure 3.4 gives an example of validation. In part (a), topactions A and B are still active. Action A has read objects x and y, and modified object x, while action B has read object x. Since *tnc* is 25, 25 read-write actions have committed since the system started. However, none of these actions invalidated either A or B; to keep the figure simple, we do not show their entries in the set *rw-committed*. Part (b) shows what happens when action A commits. Action A passes validation: as we just mentioned, there are no actions in *rw-committed* that invalidate it. A is a read-write action, so it is assigned an end time (26) after *tnc* is incremented, its shadow versions are moved up to become base versions (not shown in the figure), and a pair containing its end time and write set is added to *rw-committed*. When action B attempts to commit, it will fail validation, since it has read an out-of-date version of object x.

```
Topaction T Attempts to Commit:
    begin critical section
    st(T) := the earliest read time in rs(T)
    for each [ws(RW), et(RW)] pair in rw-committed with st(T) < et(RW) do
        for each [readobj, readtime] pair in rs(T) do
            if readobj ∈ ws(RW) and readtime < et(RW) then signal FAILED end
            end
        end
    if T is read-write then
        tnc := tnc + 1; et(T) := tnc
        install shadow versions of T as base versions
        add the pair [ws(T), et(T)] to rw-committed
        end
    (T is now a committed topaction)
    end critical section
```

Figure 3.3: Serial Validation — Basic Single-Site Model

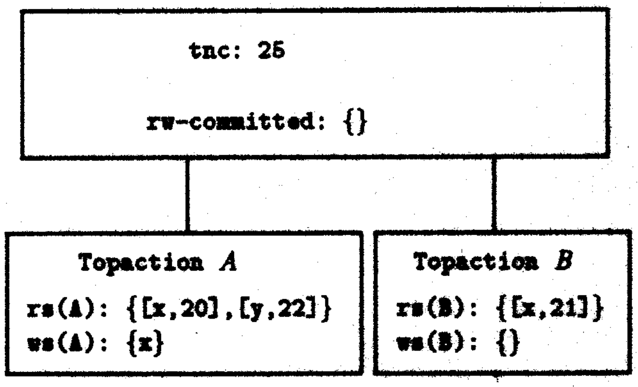
3.6 Parallel Validation

To convert our serial validation algorithm to a parallel algorithm, we introduce a validating queue, *VQ*. *VQ* contains entries for topactions that are either in their validating phase or have completed validation but have not yet installed their shadow versions. Entries in the queue can validate concurrently, but the updates of successfully validated topactions are applied in the order that the actions entered the queue: topactions are still serialized in commit order, *i.e.*, the order they entered the queue.

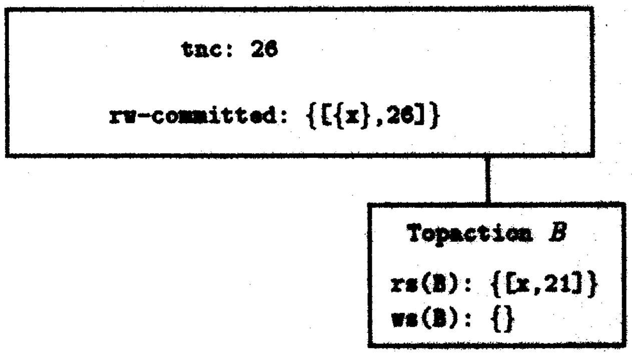
Upon entry into the queue, a topaction's entry type is set to VALIDATING. If an action passes validation, its entry type is set to READY, which means it is ready to have its updates applied; if an action fails validation, its entry is removed from the queue. Whenever the topaction at the head of the queue has type READY, it is removed and its updates are applied.

Figure 3.5 gives a pseudo-code implementation of the parallel validation algorithm. A topaction T must still validate against the committed topactions in *rw-committed*. In addition, T validates against all topactions that are in front of it in the queue. Since both the queue and *rw-committed* can change while T is validating, when T first enters the queue it takes a "snapshot" (makes copies) of *VQ* and *rw-committed*. If a topaction from either of these sets has a write set that overlaps T's read set, then T fails validation. Note that the transaction counter is incremented as a topaction's updates are installed (not when each

(a) Topactions *A* and *B* are active.



(b) Topaction *A* commits with end time = 26.



Topaction *B* will fail validation when it attempts to commit.

Figure 3.4: Validation Example — Basic Single-Site Model

read-write action enters the queue); the counter's value is used for read times, and therefore must indicate the end time of the latest committed action to install updates.

This parallel algorithm is equivalent to the serial algorithm, except for one detail. Consider the case where a topaction *T1* conflicts with a topaction *T2* that is in front of it in the queue, but *T2* has not finished validating yet. (Assume this is the only conflict for *T1*.) The algorithm given aborts *T1*; it assumes that *T2* will commit, and therefore *T1* must abort. However, if *T2* fails validation, then *T1* can commit. To be identical to the serial algorithm, the parallel algorithm would wait to see whether *T2* passed validation before deciding the validation outcome for *T1*.

There is a final synchronization issue that should be mentioned here. When a read-write commits, *tnc* is incremented and shadow versions are installed as base versions. During this update process, actions can not read base versions, *i.e.*, a base version/*tnc* update must appear to be atomic with respect to reading base versions (and recording read times). Throughout this thesis, we assume that this restriction is enforced; however, we do not show critical sections for this purpose in our figures.

3.7 Optimizations

In the descriptions above, we did not discuss when action information can be discarded. When an action aborts, all information about the action (read set, write set, start and end times, shadow versions) can be discarded. When an action commits, its write set and end time are recorded in *rw-committed*, and its shadow versions are installed as base versions; after this is done, all of the action's information can again be discarded. Finally, the set *rw-committed* can be pruned. The system can keep track of the oldest start time, OST, for any active topaction; all topaction entries with end times less than OST can be removed from *rw-committed* at any convenient time.

We do not want to scan an entire read set to determine an action's start time. Note that the first `[id(0), tnc]` pair to be entered in an action's read set contains the start time for the action; if the read set structure keeps entries in order, it will be easy to determine the start time. If the pairs are reordered (perhaps to make it easier to compare the set to write sets), then it makes sense to record the start time separately.

By choosing the right data structure, it should be possible to avoid making a copy of the queue when entering validation. For example, if the queue is represented as an array, two pointers can be used to record the scope of the queue that a particular action must

```
Topaction T Attempts to Commit:
    begin
        begin critical section
        st(T) := the earliest read time in rs(T)
        VQcopy := copy of VQ ; rw-copy := copy of rw-committed
        allocate entry E for topaction T
        E.type := VALIDATING ; add E to VQ
        end critical section
    for each [ws(RW), et(RW)] pair in rw-copy with $st(T) < et(RW)$ do
        for each [readobj, readtime] pair in rs(T) do
            if $readobj \in ws(RW)$ and $readtime < et(RW)$ then exit FAIL end
            end
        end
    for each read-write topaction RW in VQcopy do
        for each [readobj, readtime] pair in rs(T) do
            if $readobj \in ws(RW)$ then exit FAIL end
            end
        end
        begin critical section
        E.type := READY
        while head(VQ).type = READY do
            let H be the topaction at the head of VQ
            remove H's entry from VQ
            if H is read-write then
                tnc := tnc + 1; et(H) := tnc
                install shadow versions of H as base versions
                add the pair [ws(H), et(H)] to rw-committed
                end
            (H is now a committed topaction.)
            end
        end critical section
    end
        except when FAIL:
                    begin critical section
                    delete E from VQ
                    exit critical section
                signal FAILED
                end
```

Figure 3.5: Parallel Validation — Basic Single-Site Model

validate against. Similarly, we should be able to avoid copying *rw-committed*.

The performance of the validation queue can be improved by allowing READY topactions to move ahead of VALIDATING topactions. A topaction can move ahead of another topaction if its write set does not overlap the read set of the action that it is passing. As a special case of this queue movement, a read-only action can be placed at the front of the queue when it enters validation; *i.e.*, it can be moved in front of all other actions.

Action reordering in the validation queue can also be used to reduce the number of actions that fail validation: an action that conflicts with another action that is in front of it in the queue might be able to move in front of that action; in this case, it no longer has to validate against the conflicting action. This idea is explored in [Agrawal *et al.* 1987].

Chapter 4

Single-Site Nested Action Model

In the basic single-site model, our main contribution was to introduce a read time for each object in an action's read set (as opposed to using the start time of an action as the read time for each object in the read set). This should reduce the number of unnecessary aborts.

Beginning with this chapter, we present original work. In the topaction-only model, we used shadow versions to provide the recoverability property of actions, and we used validation algorithms (based on read-write conflict detection) to provide the serializability property. In this chapter, we extend the use of shadow versions to provide the recoverability property for subactions, and, starting with a definition of serializability for nested actions, we develop several algorithms for validating nested actions in a single-site system. The result is a centralized system with nested actions. For an overview of nested actions, see Chapter 2 and Moss's Ph.D. thesis [Moss 1981].

As we did with the basic single-site model, we present both serial and parallel validation algorithms. First, we give a single serial algorithm that can be used to serialize all actions. Only one action (topaction or subaction) is validated at a time. We then add some parallelism by introducing a validation queue. All actions in the system are placed in this queue, and each action in the queue validates against any of its siblings that are in front of it in the queue. Placing topactions and subactions in the same queue tends to slow down subaction processing; we also present a second parallel validation algorithm that uses two queues, one for subactions and one for topactions.

Finally, we generalize our approach by showing that it is possible to partition the set of subactions into disjoint sets, where one validation queue can used for the validation of each set. Additional validation queues introduce additional concurrency, which is useful on a multiprocessor (or in a distributed system).

4.1 Shadow Versions and Nested Actions

Recall that in the basic single-site model, when a topaction modifies an object, a shadow version of the object is created. We say that this shadow version *shadows* the base version of the object, which is stored in stable storage; the topaction reads and writes the shadow version rather than the base version. If a topaction aborts, its shadow versions are thrown away, and the topaction has no effect on the base versions of the objects; if a topaction commits, its shadow versions are installed as base versions in one atomic step. The result is that topactions have recoverability.

Shadow versions can be used in the same way to provide recoverability for subactions. When a subaction reads an object, it reads the version of the object located at the nearest ancestor that has a shadow version. In other words, a subaction reading object O first checks to see whether it has its own shadow version of O; if it does not, it checks to see whether its parent has a shadow version of O; if not, it checks its parent's parent; and so on. If no ancestor has a shadow version, then it reads the base version of the object.

For convenience, we say that stable storage is the "parent" of all topactions in the system. This allows us to formulate a general rule for any action: when reading an object O, the version located at the nearest ancestor that has a version is read. (From now on, we will simply say that the *nearest version* is read.)

We can also give a general rule for writing an object: when an action A is about to write an object O for the first time, a shadow version of the object is created for A, based on the nearest version. All modifications made by A to O are made to this shadow version. Thus, we can have shadows of shadows, and shadows of shadows of shadows, and so on.

If an action aborts, its shadow versions are discarded, and it has no effect on its parent action's versions. If an action commits, its shadow versions are moved up to become versions of its parent, possibly replacing versions already located at the parent. Note that this rule works for topactions: when a topaction commits, its shadow versions are moved up to its parent (stable storage) to become base versions.

After a subaction commits, its parent action might abort, causing the shadow versions that were moved up to the parent to be thrown away. The modifications made by a committed subaction only become permanent if all of its ancestors commit; *i.e.*, if the subaction commits through the top. If an ancestor of a committed subaction aborts, we say that the effects of the subaction are *undone*.

Figure 4.1 gives an example of the use of shadow versions for subactions. Part (a) shows

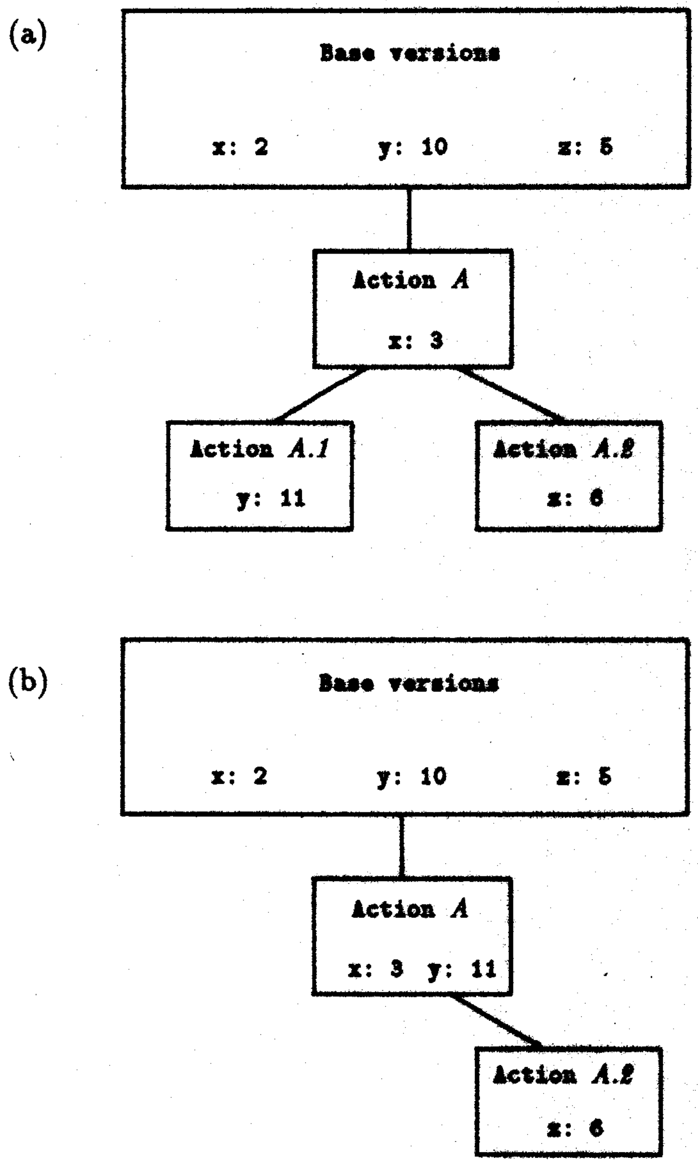


Figure 4.1: Shadow Versions — Nested Single-Site Model

a system with three objects, x, y, and z, and a topaction A that has modified object x and started two concurrent subactions, *A.1* and *A.2*. *A.1* has modified object y, while *A.2* has modified z. Note that if *A.1* reads object y, it will read its own version, while if *A.2* reads y it will read A's version of y. Part (b) shows the system after *A.1* commits: its shadow version for y has been moved up to its parent, A. Note that this update will not become permanent unless A commits: if A commits, its shadow versions will be moved up to become base versions, while if it aborts its shadow versions will be thrown away.

4.2 Defining Serializability for Subactions

Validation is used to ensure the serializability of actions. For a system with topactions only, this property can be defined as follows: *the net effect of all committed topactions is the same as the net effect of executing those topactions in some non-overlapping order.*

What is the corresponding property for a system with both topactions and subactions? The answer is that each set of *sibling actions* is serialized (where all topactions are considered siblings). The property can be stated as follows: *for each set of siblings S, the net effect of executing all actions in S that commit is the same as the net effect of executing those actions in some non-overlapping order.*

The net effect of executing an action A includes the effects of all descendants that have committed up to A. Thus, if *S1* and *S2* are committed siblings, with *S1* ordered before *S2*, then any action committed up to *S1* is ordered before *S2* and any action committed up to *S2*. In other words, an ordering on siblings also gives an ordering for all actions in the system that commit through the top. Actually, a parent action is not ordered with respect to its children. In practice, this is not an issue, since a parent blocks while its children run, and it is easy to reason about the interaction between parent and child. Formal models of nested actions consider each access to an object (*e.g.*, each read or write) to run as a subaction. With this addition, ancestors are ordered after descendants [Lynch *et al.* 1988].

4.3 Overview of Validation

As we mentioned above, we serialize each set of sibling actions (where the topactions are a set of siblings). Validation for each set of siblings is essentially equivalent topaction validation in the basic model. Sibling actions are serialized in the order that they enter validation: we only allow an action to commit if it can be serialized *after* all of its siblings that have already committed.

```
Function nearest-version(A, id(O)):
    if shadow(A, id(O)) exists then return shadow(A, id(O)) end
    if A is a topaction then
            return base-version(id(O))
        else
            return nearest-version(parent-of(A), id(O))
        end
```

Figure 4.2: The `nearest-version` Function — Nested Single-Site Model

We begin by presenting a serial algorithm; we then modify it as necessary to produce a single-queue parallel algorithm (one queue for all actions) and a double-queue parallel algorithm (one queue for topactions and one queue for subactions).

4.4 Data Structures

There is a single monotonically increasing transaction counter, *tnc*. Each action *A* (topaction or subaction) has a read set *rs(A)* and a write set *ws(A)*, as in with the basic model. Because shadow version management is more complicated, we introduce formal data structures for shadow versions and base versions: *shadow(A, id(O))* gives *A*'s shadow version for *O*, if it exists, and *base-version(id(O))* gives the base version for object *O*. The function **`nearest-version`** returns the nearest version of an object (see Figure 4.2).

4.5 Read and Write

The rules for reading and writing an object are shown in Figure 4.3. They are similar to the topaction-only operations from Figure 3.2, except for the use of the nearest-version function.

4.6 Commit-Time Update

When a subaction commits, the update phase is responsible for installing the effects of the subaction at its parent. As we have already mentioned, one aspect of installing its effects is to transfer its shadow versions to its parent. In addition, the subaction's read set and write set must be transferred to the parent, since the result of the commit is that the parent has adopted the reads and writes of the child.

The commit-time update process for subactions and topactions is shown in Figure 4.4.

```
Action A reads Object O:
    if id(O) ∉ rs(A) then add the pair [id(O), tnc] to rs(A) end
    read nearest-version(A, id(O))

Action A writes Object O:
    if id(O) ∉ ws(A) then begin
        add id(O) to ws(A)
        shadow(A, id(O)) := copy(nearest-version(A, id(O))
        end
    modify shadow(A, id(O))
```

Figure 4.3: Read and Write — Nested Single-Site Model

```
Commit Action C:

    If C is a Topaction:
        As one atomic step:
            for each id(O) in ws(C) do
                base-version(id(O)) := shadow(C, id(O))
                end

    If C is a Subaction with Parent P:
        for each id(O) in ws(C) do
            shadow(P, id(O)) := shadow(C, id(O))
            end
        for each [id(O), readtime-C] in rs(C) do
            if there is an [id(O), readtime-P] in rs(P) then
                    replace [id(O), readtime-P] with
                            [id(O), min(readtime-C, readtime-P)] in rs(P)
                    end
                else
                    add [id(O), readtime-C] to rs(P)
                end
            end
        for each id(O) in ws(C) do
            if id(O) ∉ ws(P) then add id(O) to ws(P) end
            end
```

Figure 4.4: Commit-Time Update — Nested Single-Site Model

```
Action C with Parent P Attempts to Commit:
    begin critical section
    st(C) := the earliest read time in rs(C)
    for each [ws(RW), et(RW)] pair in rw-committed(P) with st(C) < et(RW) do
        for each [readobj, readtime] pair in rs(C) do
            if readobj ∈ ws(RW) and readtime < et(RW) then signal FAILED end
            end
        end
    if C is read-write then
        tnc := tnc + 1; et(C) := tnc
        add the pair [ws(C), et(C)] to rw-committed(P)
        end
    commit action C
    (C is now a committed action)
    end critical section
```

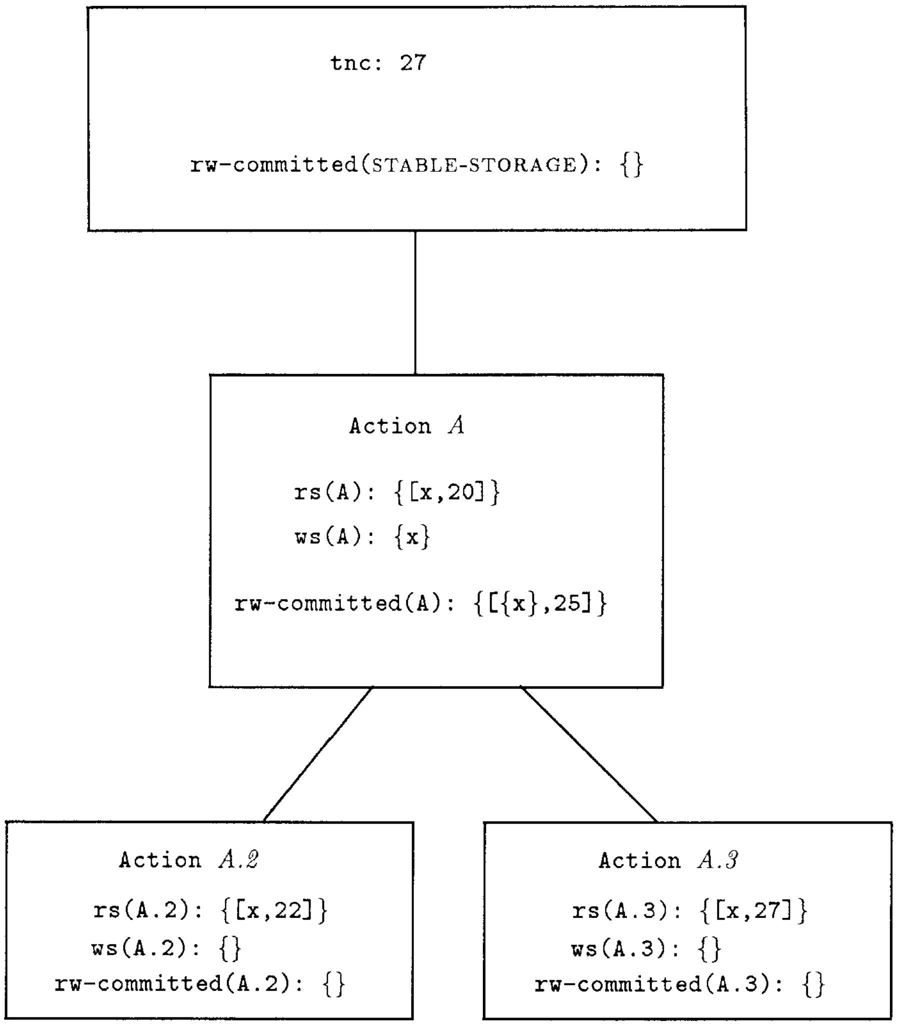
Figure 4.5: Serial Validation — Nested Single-Site Model

For a topaction, we install the action's shadow versions as base versions. For a subaction, we move the action's shadow versions to its parent and merge its read and write sets into its parent's sets. Since write sets are sets of object identifiers, the parent's new write set is simply the union of its current write set and the subaction's write set. Read sets, on the other hand, are sets of pairs of the form `[id(O), readtime]`. We take the union of the parent's read set and the subaction's read set, with the following extra rule. If both the sets have a pair with the same identifier, we keep only one of these pairs in the union — the pair with the *earlier* read time.

4.7 Serial Validation

Figure 4.5 gives pseudo-code for the validation of any action, subaction or topaction. Each set of siblings has its own set of committed read-write actions: if P is the parent of C, then *rw-committed(P)* gives the committed read-write siblings of C. Thus, *rw-committed(P)* is the read-write committed set for all of P's children. If C is a topaction, then its parent is the special value STABLE-STORAGE, and *rw-committed(*STABLE-STORAGE*)* gives the set of committed read-write topactions. This algorithm is very similar to the serial algorithm that was presented for the basic model (Figure 3.3).

Figure 4.6 gives a validation example. Topaction A has started three subactions, *A.1*, *A.2*, and *A.3*, where *A.1* has already committed. We can see from the set *rw-committed(A)*



Action *A.1* committed at time 25, updating object x. Action *A.2* will fail validation, while action *A.3* will pass.

Figure 4.6: Validation Example — Nested Single-Site Model

that action *A.1* ended at time 25 and updated object x. Concurrent siblings *A.2* and *A.3* have both read x, where *A.2* read x before time 25 and *A.3* read x after time 25. Thus, *A.2* will fail validation, while *A.3* will pass.

4.8 Parallel Validation

In this section we present several parallel validation algorithms. The first algorithm uses a single validation queue for both subactions and topactions. The second algorithm uses two queues, one for subactions and one for topactions. Finally, we discuss the possibility of introducing additional queues.

4.8.1 Parallel Validation: One Queue

A parallel validation algorithm is given in Figure 4.7. The algorithm works for both subactions and topactions. It is similar to the algorithm from Figure 3.5: it introduces a single validation queue, *VQ*, where all actions enter at validation time. Queue entries are marked as either VALIDATING or READY. Entries for actions that fail validation are removed. At the front of the queue, READY actions are removed, and their updates are applied.

A given action C validates against two sets of siblings: read-write siblings that have already committed, as recorded in *rw-committed(P)* (where P is the parent of C), and read-write siblings that are in *VQ* at the time that C enters the queue. Because *rw-committed(P)* and *VQ* can both change while C is validating, a "snapshot" of the two sets is taken at the time that C enters the queue. A complete copy of *rw-committed(P)* is made, and a copy of *VQ* is made that only records entries for read-write siblings of C. (In the code, the call `rw-sibling-copy(VQ,C)` performs this selective copy.)

4.8.2 Parallel Validation: Two Queues

While actions can validate in parallel using the above algorithm, their updates are applied serially, as READY entries are removed from the front of the queue. For the topaction-only case, this was reasonable, since all updates were to stable storage. However, for this model there is a problem: updates for topactions (to stable storage) take considerably longer than updates for subactions. If a subaction finishes validating, but a topaction that is ahead of it is READY, the subaction will have to wait until the topaction finishes updating before applying its own updates.

```
Action C with Parent P Attempts to Commit:
    begin
        begin critical section
        VQcopy := rw-sibling-copy(VQ,C) ; rw-copy := copy(rw-committed(P))
        allocate entry E for action C
        E.type := VALIDATING; add E to VQ
        end critical section
    st(C) := earliest read time in rs(C)
    for each [ws(RW), et(RW)] pair in rw-copy with $st(C) < et(RW)$ do
        for each [readobj, readtime] pair in rs(C) do
            if readobj $\in$ ws(RW) and readtime $<$ et(RW) then exit FAIL end
            end
        end
    for each read-write action RW in VQcopy do
        for each [readobj, readtime] pair in rs(C) do
            if readobj $\in$ ws(RW) then exit FAIL end
            end
        end
        begin critical section
        E.type := READY
        while head(VQ).type = READY do
            let H be the action at the head of VQ
            remove H's entry from VQ
            if H is read-write then
                tnc := tnc + 1; et(H) := tnc
                add the pair [ws(H), et(H)] to rw-committed(P)
                end
            commit action H
            (H is now a committed action.)
            end
        end critical section
    end
        except when FAIL:
                begin critical section
                delete E from VQ
                end critical section
            signal FAILED
            end
```

Figure 4.7: Single-Queue Parallel Validation — Nested Single-Site Model

Since subactions do not validate against topactions, it is actually possible to move subactions forward over topactions in the queue. However, a subaction can not be moved in front of a topaction that is currently installing its updates, since the update process occurs in a critical section. In addition, subactions can not enter the queue during a critical section. Thus, the relative slowness of topaction updates *will* slow down the processing of subactions.

One solution is to introduce two validation queues, one for topactions and one for subactions. This allows subactions to be processed at a faster rate. Since each queue has its own critical section, fewer actions use a given critical section: this will speed up the processing of both topactions and subactions. (In general, we can split up the actions in the system into N sets, and use N validation queues, thus speeding up validation for all N sets. This idea will be discussed later.)

Using two queues is not as straightforward as it might seem: it is no longer possible to use a single transaction counter. The transaction counter records the end time of the last committed read-write action to have its updates applied. With two queues concurrently making updates (topaction updates and subaction updates) we need two such counters. Counter *tnc-top* records the end time of the last topaction to have its updates applied, and counter *tnc-sub* records the end time of the last subaction to have its updates applied.

We must now record both counter values when recording subaction read times. Suppose an subaction reads an object O. When it validates, its read time for O must be compared to the end times of committed siblings; thus, the *tnc-sub* value must be recorded. If this subaction commits up to a topaction, and the topaction then commits, then the subaction's read time for O (inherited by the topaction) must be compared to the end times of committed topactions; thus, the *tnc-top* value must be recorded. As a result, we now record the triple `[id(O), tnc-sub, tnc-top]` when an action reads object O. Note that, when merging a child's read set into its parent's read set, as we do in Figure 4.4, if both the child and parent had a triple for object O, the earlier *tnc-sub* and *tnc-top* times are retained.

Figure 4.8 gives the new validation algorithm for subactions. It uses subaction queue *VQ-SUB* for adding subaction entries, counter *tnc-sub* for subaction end times, the middle (*tnc-sub*) entry from read set triples for testing read times, and a critical section for subactions. The topaction algorithm is identical, except that it uses topaction queue *VQ-TOP*, counter *tnc-top*, the third (*tnc-top*) entry from read set triples, and a critical section for topactions.

```
Subaction C with Parent P Attempts to Commit:
  begin
     begin critical section for subactions
     VQcopy := rw-sibling-copy(VQ-SUB, C) ; rw-copy := copy(rw-committed(P))
     allocate entry E for action C
     E.type := VALIDATING; add E to VQ-SUB
     end critical section for subactions
  st(C) := the earliest tnc-sub read time in rs(C)
  for each [ws(RW), et(RW)] pair in rw-copy with st(C) < et(RW) do
     for each [readobj, readtime-sub, readtime-top] triple in rs(C) do
        if readobj ∈ ws(RW) and readtime-sub < et(RW) then exit FAIL end
        end
     end
  for each read-write action RW in VQcopy do
     for each [readobj, readtime-sub, readtime-top] triple in rs(C) do
        if readobj ∈ ws(RW) then exit FAIL end
        end
     end
     begin critical section for subactions
     E.type := READY
     while head(VQ-SUB).type = READY do
        let H be the action at the head of VQ-SUB
        remove H's entry from VQ-SUB
        if H is read-write then
           tnc-sub := tnc-sub + 1; et(H) := tnc-sub
           add the pair [ws(H), et(H)] to rw-committed(P)
           end
        commit action H
        (H is now a committed action.)
        end
     end critical section for subactions
  end
     except when FAIL:
              begin critical section for subactions
              delete E from VQ-SUB
              end critical section for subactions
           signal FAILED
           end
```

Figure 4.8: Two-Queue Parallel Validation — Nested Single-Site Model

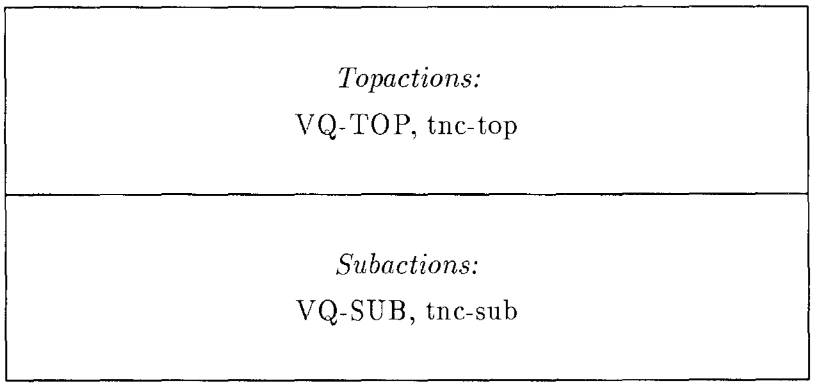


Figure 4.9: A Horizontal Partition Between Topactions and Subactions

4.8.3 Additional Validation Queues

At this point, it should be clear that one can use any number of validating queues, not just one or two. The actions in the system can be partitioned into any number of disjoint sets, and one validation queue can be used for each set. Since an action must validate against its siblings, only partitions that do not separate siblings are acceptable. For the two-queue case, we say that there is a *horizontal partition* between the set of topactions and the set of subactions; this is depicted in Figure 4.9.

We can make further horizontal partitions. A complete horizontal partitioning would have one queue for the topactions, one for the children of topactions, one for the children of the children of topactions, and so on, as depicted in Figure 4.10.

While introducing more queues improves the performance of each queue, there is a reason to avoid introducing horizontal partitions: for N horizontal sets, we need N-part timestamps for read times. For example, for two horizontal sets (topactions and subactions) read times are two-part timestamps. This is needed because each queue validates a read time by comparing it to the end times of committed read-write actions that were validated by that queue. As an item in a read set moves up the action tree, each partition's queue validates the read time. Thus, for the complete horizontal partition above, when an action at level k in the action tree reads an object, it needs to record a k-part timestamp, one part for each queue that might validate the read. A topaction that reads an object only needs to record a single *tnc* value, a child of a topaction would record two *tnc* values, and so on. (As read sets entries move up an action tree, k-part timestamps can be truncated to

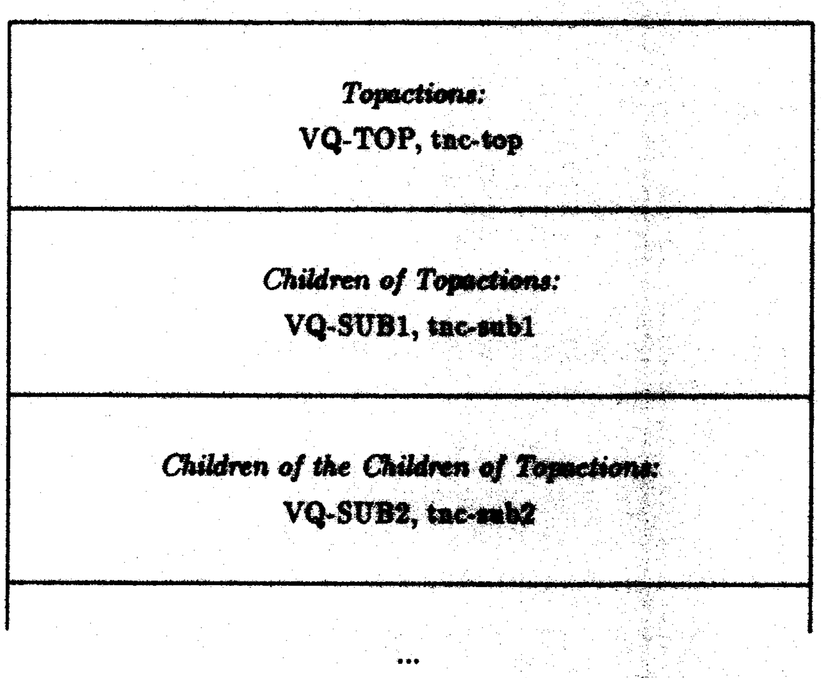

Topactions:
VQ-TOP, tnc-top
Children of Topactions:
VQ-SUB1, tnc-sub1
Children of the Children of Topactions:
VQ-SUB2, tnc-sub2
...

Figure 4.10: A Complete Horizontal Partitioning

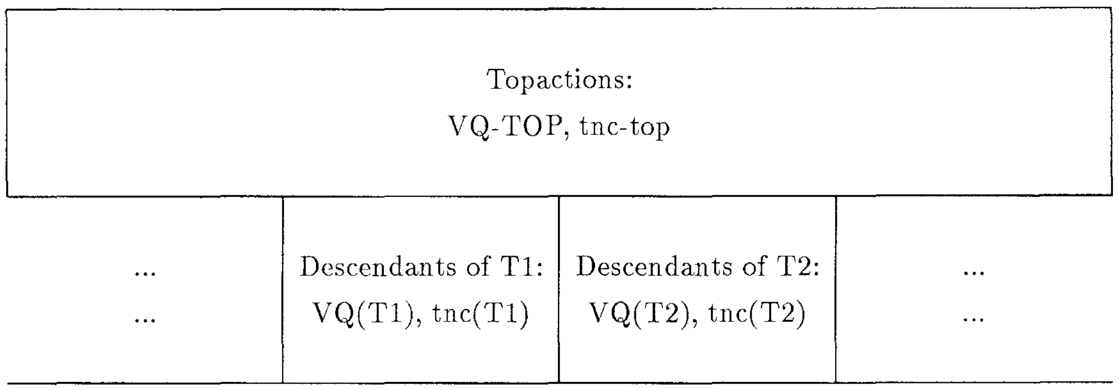


Figure 4.11: A Vertical Partitioning of the Subactions

$(k-1)$-part timestamps.)

It is also possible to create *vertical partitions*. For example, returning to our original horizontal partition (topactions and subactions), we can partition the set of subactions into disjoint sets, one for the descendants of each topaction in the system; this is depicted in Figure 4.11.

Here, there is a subaction queue and corresponding counter for each topaction in the system. All descendants of topaction *T1* will be placed in subaction queue *VQ(T1)* for validation, and will take end times from *tnc(T1)*. Since there is still only one horizontal partition, read times will still be two-part timestamps. A descendant of topaction *T1* records *tnc(T1)* and *tnc-top* in its read set entry: the *tnc(T1)* value is used to validate it with respect to other descendants of topaction *T1*, while the *tnc-top* value is used to validate topaction *T1* with respect to other topactions.

A horizontal partition between the topactions and the subactions is always useful: topaction processing involves stable storage updates, while subaction processing does not. Further horizontal partioning is probably not useful, since the size of a read-time timestamp grows with the number of horizontal partitions. Using a vertical partition for subactions is probably only useful on a multiprocessor. Additional queues allow us to apply multiple subaction updates concurrently; however, on a uniprocessor, only one subaction update can be processed at a time.

Our two distributed models both use partitions. The fixed action model uses a single horizontal partition, between topactions and subactions, while the fixed object model uses a complete horizontal partioning. Thus, the fixed action model uses 2-part timestamps for read times, while the fixed object model uses multi-part timestamps. Although we just stated that we would like to avoid a complete horizontal partition, we can not avoid using one in the fixed object model. Both models use vertical partitioning within each horizontal set of subactions.

4.9 Optimizations

In the topaction-only parallel validation algorithm (Figure 3.5), a READY entry can switch places with a VALIDATING entry in front of it in the queue as long as the read set of the VALIDATING entry does not overlap the write set of the READY entry. For both the one-queue or two-queue algorithms presented above, this condition still holds for sibling entries. In addition, any two *non-sibling* entries can always be switched, since they do not validate against each other. This switching process could take place as part of the second critical section of the algorithms: when an action sets its entry type to READY, it could push itself as far forward in the queue as possible before performing the loop that removes READY entries from the front of the queue. (For the one-queue algorithm, however, READY topactions should not push beyond READY subactions.)

Once again, action information can be discarded. When an action commits or aborts, all information about the action can be thrown away. (For a commit, information that must be kept is recorded elsewhere: shadow versions are moved, an entry is added to the parent's *rw-committed* set, and read set and write set information is inherited by the parent.) Note that all *rw-committed* sets will be thrown away except one — the set for topactions. The topaction set can be pruned as we described in the last chapter.

Chapter 5

Fixed Action Model

This chapter and the following one present our two models for nested distributed transactions. They present simple abstract models for the validation of subactions and topactions in a distributed system. We hope that they demonstrate that real implementations of the models are feasible; however, they do not present the most efficient ways to build such implementations. Having said this, there *are* some optimizations that can be presented without discussing low-level implementation details. We discuss these optimizations in the final section of each chapter.

For the single-site model described in Chapter 4, we assumed that there was only one entity in the system. This entity managed all of the objects in the system and also managed all of the actions. In a distributed system, we would like to have any number of entities: some manage objects and some manage actions.

This chapter presents the first of our two distributed models, the *fixed action model.* The fixed action model has the constraint that a topaction and all of its descendants must run at a single entity. This entity need not be an object manager (OM). In fact, we introduce a new entity type for managing actions, an *action manager* (AM). Actions run at AM's, whereas objects are managed by OM's. An action running at an AM can read or write objects managed by any of the OM's in the system; if an action needs to read the base version of an object, the action's AM sends a `base-version-request` message to the object's OM, and the OM returns a copy of the base version. All shadow versions for a given topaction and its descendants are located at the topaction's AM, while the base versions accessed by the topaction are located at any number of OM's. Figure 5.1 gives an example of a set of AM's and OM's.

The serial validation algorithms presented in the previous two chapters were useful for demonstration purposes, but are too inefficient to be practical. For our distributed models,

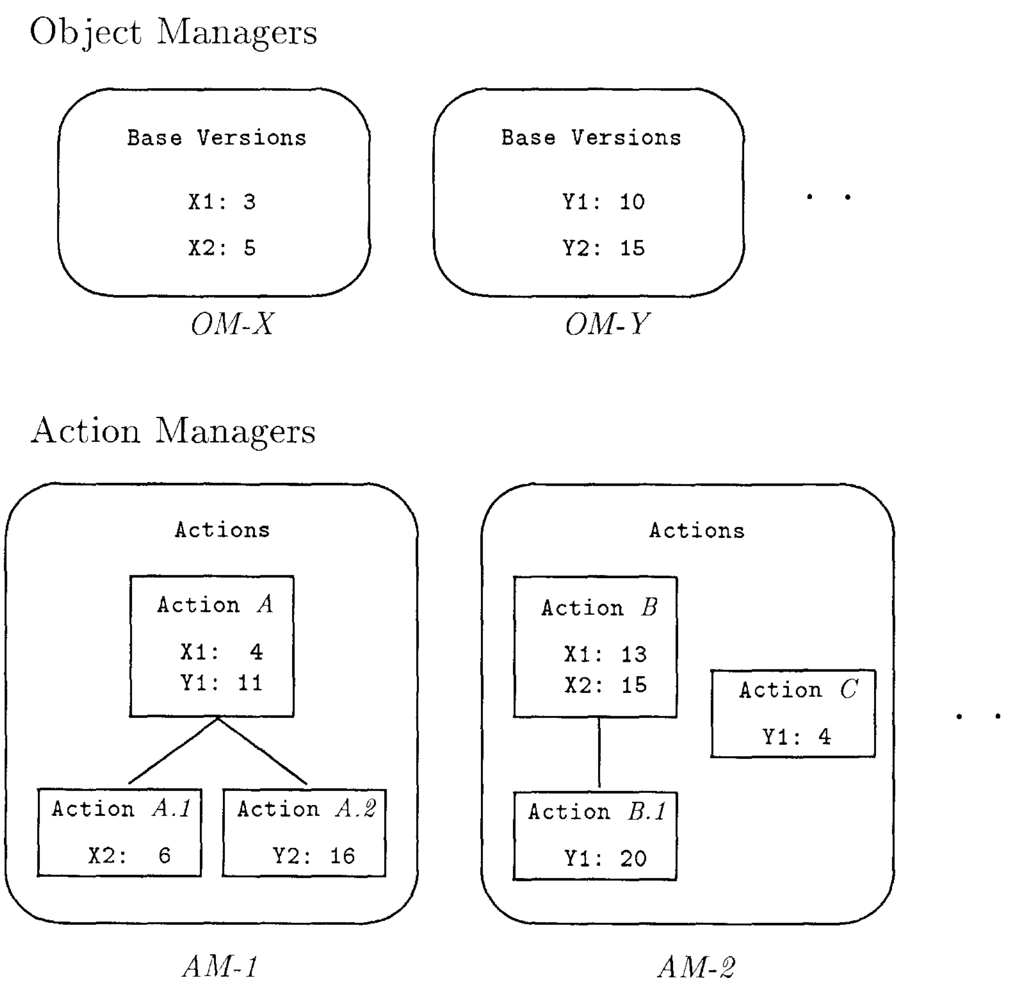


Figure 5.1: Object Managers and Action Managers — Fixed Action Model

we only describe parallel algorithms. Validation in the fixed action model is similar to the two-queue parallel algorithm from the last chapter (see Section 4.8.2). Each AM has a single queue for validating subactions that run at the AM, and each OM has a single queue for validating topactions that have requested base versions from that OM. Although there are multiple topaction queues in the system (one at each OM), the topaction validation process is synchronized; one can think of the OM's in the system as implementing a single topaction queue. The multiple subaction queues (one at each AM) do not have to be synchronized. There is a vertical partition of the subactions in the system, where the subactions that run at a given AM are placed in the same set. (If a subaction runs at an AM, all of its siblings also run at that AM. Since all descendants of a topaction run at the same AM, we could use one subaction queue per topaction instead of one per AM; this extra parallelism would only be useful on a multiprocessor.)

5.1 Action Managers and Object Managers

Recall that our abstract model for a distributed system (described in Chapter 2) is a set of communicating entities. The fixed action model has two types of entities, object managers (OM's) and action managers (AM's).

Each entity has a globally unique identifier. An OM manages a set of objects whose base versions are kept in stable storage. A given object *O* at an object manager *OM-X* has a globally unique identifier, *id(O)*, that includes the identifier of *O*'s object manager, *id(OM-X)*. Thus, if an action has an object identifier for an object that it wants to read, the identifier tells the action's AM which OM it should contact to request a copy of the base version.

Each AM manages some topactions and their descendants. OM's manage the base versions of objects, while AM's manage all shadow versions of objects. When an action reads an object, it still reads the nearest version; however, if no ancestor has a version, the action's AM gets a copy of the base version by sending a `base-version-request` message to the object's OM and waiting for the response.

5.2 Data Structures at an Action Manager

All actions *A* have a read set *rs(A)*, a write set *ws(A)*, and a set *rw-committed(A)* that contains information about the committed read-write children of *A*. At each action manager, there is a subaction validation queue *VQ-sub* and a transaction counter *tnc-sub*. Each AM also has its own shadow version data structure: *shadow(A, id(O))* gives *A*'s shadow version for *O*, if the shadow version exists.

Figure 5.2 shows the `nearest-version` function for this model. An action's ancestors are all located at the same AM as the action, but the base versions of objects are located at the various OM's in the system — reading the nearest version of an object often involves sending a message to an object manager. Compare this to Figure 4.2, where the base version is accessed directly. The OM returns a response that includes two values: a copy of the base version of *O*, and the value of its transaction counter. The OM's counter is recorded as part of the read time for the object; when the topaction commits, this read time will be sent back to the OM so that it can validate the read.

Topactions record read set entries of the form `[id(O), OM-readtime]`, while subactions record read set entries of the form `[id(O), tnc-sub, OM-readtime]`. Note that the `nearest-version` function returns both an object and the appropriate value for *OM-readtime*, which is recorded as part of a topaction or subaction read set entry. When a topaction calls `nearest-version` to read an object for the first time, the *OM-readtime* returned will always be the result of a `base-version-request` message. When a subaction calls `nearest-version` to read an object for the first time, the *OM-readtime* returned will be from one of two places. If a base version is read, `nearest-version` returns the *OM-readtime* from the result of the `base-version-request`; if a shadow version from an ancestor is read, then the function returns the *OM-readtime* that the ancestor recorded.

5.3 Data Structures at an Object Manager

Object managers manage the base versions of objects. At an OM, *base-version(id(O))* gives the base version of an object *O* that the OM manages. Object managers are also responsible for the validation of topactions. If a topaction requests one or more base versions from object manager *OM-X*, we say that *OM-X* is a *participant* in the topaction.

Each OM has a validation queue *VQ-TOP*, a transaction counter *tnc-top*, a queue entry time counter *qet*, and a set *rw-top-committed*. The use of the new counter *qet* is described in Section 5.6, where we describe topaction validation. The set *rw-top-committed* contains information about committed read-write topactions in which the OM participated.

```
Function nearest-version(A, id(O)):
    if shadow(A, id(O)) exists then
        if A is a topaction then
            look up the tuple [id(O), OM-readtime] in rs(A)
          else
            look up the tuple [id(O), sub-readtime, OM-readtime] in rs(A)
          end
        return two values: shadow(A, id(O)) and OM-readtime
      elseif A is a topaction then
        OM-O := O's object manager
        send the message base-version-request(id(O)) to OM-O
        return the result of this message
        (the result will be a copy of O's base version
         and the value of OM-O's transaction counter)
      else
        return nearest-version(parent-of(A), id(O))
      end
```

Figure 5.2: The `nearest-version` Function — Fixed Action Model

5.4 Read and Write

The rules for reading and writing an object (which only occurs at action managers) are shown in Figure 5.3. They are essentially the same as the single-site rules, except that the function `nearest-version` returns both an object and a read time, as we just described. Reading an object at an AM can cause a `base-version-request` to be sent to an OM: OM processing of a `base-version-request` message for object O is described in Figure 5.4. The current value of *tnc-top* is returned along with the value of O's base version; this *tnc-top* value is recorded by the requesting action as part of the read set entry for O.

5.5 Subaction Commit Process

Each set of sibling subactions runs at a single action manager; each action keeps the same information that actions kept in the single-site model. Subaction validation for the fixed action model is no different from subaction validation for the single-site model.

5.5.1 Commit-Time Update

When a subaction commits, its read set, write set, and shadow versions are moved up to its parent; this is shown in Figure 5.5. The figure is similar to Figure 4.4 in the last chapter;

```
Action A reads Object O:
    obj, OM-readtime := nearest-version(A, id(O))
    if id(O) $\notin$ rs(A) then
        if A is a topaction then
                add the pair [id(O), OM-readtime] to rs(A)
            else
                add the triple [id(O), tnc-sub, OM-readtime] to rs(A)
            end
        end
    read obj

Action A writes Object O:
    if id(O) $\notin$ ws(A) then
        obj, OM-readtime := nearest-version(A, id(O))
        add id(O) to ws(A)
        shadow(A, id(O)) := copy(obj)
        end
    modify shadow(A, id(O))
```

Figure 5.3: Read and Write — Fixed Action Model

```
OM receives message: base-version-request(id(O))
    begin
    return two values: base-version(id(O)) and tnc-top
    end
```

Figure 5.4: Processing a `base-version-request` Message — Fixed Action Model

the only difference is that subaction read set entries have two-part timestamps, so that the merging of read sets is different. If the parent of a committing subaction is a topaction, then the two-part timestamps are truncated to one-part timestamps: the *tnc-sub* read times are removed.

5.5.2 Validation

Each action manager has a single queue *VQ-SUB* for validating subactions. Figure 5.6 gives the validation algorithm for this queue; it is identical to Figure 4.8, the parallel validation algorithm for the two-queue approach described in the last chapter. The function call `rw-sibling-copy(VQ-SUB,C)` is a selective copy of *VQ-SUB*, where only the read-write siblings of *C* are copied.

```
Commit Subaction C (to Parent P):
    for each id(O) in ws(C) do
        shadow(P, id(O)) := shadow(C, id(O))
        end
    if P is a topaction then
        for each [id(O), C-rt-sub, C-rt-top] in rs(C) do
            if there is an [id(O), P-rt-top] in rs(P) then
                    replace [id(O), P-rt-top] with
                            [id(O), min(C-rt-top, P-rt-top)] in rs(P)
                else
                    add [id(O), C-rt-top] to rs(P)
                end
            end
        end
    if P is a subaction then
        for each [id(O), C-rt-sub, C-rt-top] in rs(C) do
            if there is an [id(O), P-rt-sub, P-rt-top] in rs(P) then
                    replace [id(O), P-rt-sub, P-rt-top] with
                            [id(O), min(C-rt-sub, P-rt-sub), min(C-rt-top, P-rt-top)] in rs(P)
                else
                    add [id(O), C-rt-sub, C-rt-top] to rs(P)
                end
            end
        end
    for each id(O) in ws(C) do
        if id(O) is not in ws(P) then add id(O) to ws(P) end
        end
```

Figure 5.5: Subaction Commit-Time Update -- Fixed Action Model

```
Subaction C with Parent P Attempts to Commit:
    begin
        begin critical section
        VQcopy := rw-sibling-copy(VQ-SUB, C) ; rw-copy := copy(rw-committed(P))
        allocate entry E for action C
        E.type := VALIDATING; add E to VQ-SUB
        end critical section
    st(C) := the earliest tnc-sub read time in rs(C)
    for each [ws(RW), et(RW)] pair in rw-copy with st(C) < et(RW) do
        for each [readobj, readtime-sub, readtime-top] triple in rs(C) do
            if readobj ∈ ws(RW) and readtime-sub < et(RW) then exit FAIL end
            end
        end
    for each read-write action RW in VQcopy do
        for each [readobj, readtime-sub, readtime-top] triple in rs(C) do
            if readobj ∈ ws(RW) then exit FAIL end
            end
        end
        begin critical section
        E.type := READY
        while head(VQ-SUB).type = READY do
            let H be the action at the head of VQ-SUB
            remove H's entry from VQ-SUB
            if H is read-write then
                tnc-sub := tnc-sub + 1; et(H) := tnc-sub
                add the pair [ws(H), et(H)] to rw-committed(P)
                end
            commit action H
            (H is now a committed action.)
            end
        end critical section
    end
        except when FAIL:
                begin critical section
                delete E from VQ-SUB
                end critical section
            signal FAILED
            end
```

Figure 5.6: Subaction Validation — Fixed Action Model

5.6 Topaction Commit Process

Unlike a subaction, a topaction can not be validated locally by its AM. A topaction running at one AM might conflict with a committed topaction that ran at some other AM. If two topactions have a read-write conflict over some object O, then O's OM will be a participant in both topactions; we can use the object managers to validate topactions. Each OM is responsible for detecting topaction conflicts over any of its own objects; topaction validation is a distributed process, where each participating OM validates the topaction with respect to its own objects. The validation that occurs at each OM is described below, after we describe the overall distributed process.

When a topaction T attempts to commit, each participant in T is asked to validate T: the topaction's AM sends `validate-request` messages to all participants, and waits for the responses. The topaction is allowed to commit only if every participant OK's the commit. If T is allowed to commit, T's AM sends a `commit` message to all participants in T. Commit messages include the shadow versions that T has updated, so that the participants can install these shadow versions as new base versions.

5.6.1 Two-Phase Commit

We must ensure that a topaction either commits or aborts at all participants; *i.e.*, we must ensure that its updates are either installed at all of the read-write participants or at none of them. A two-phase commit protocol [Gray 1979] is used for this purpose. The details of this protocol can be found elsewhere [Gray 1979, Mohan & Lindsay 1983, Lampson 1981, Lindsay *et al.* 1979, Lindsay *et al.* 1984]; we omit many of for details here. For a pessimistic system, the first phase of the two-phase commit is the *prepare* phase, where all participants must agree that they are prepared to commit. For an optimistic system, the first phase is a combined *validate and prepare* phase, where all participants must agree that the topaction has been successfully validated and that they are prepared to commit. In other words, we send `validate-request` messages instead of prepare messages during phase one of the two-phase commit.

The topaction's AM acts as coordinator of the two-phase commit. In the first phase, the coordinator sends a `validate-request` message to each participant, and waits for responses. Participants respond with either OK or FAILED. If any participant responds with FAILED, then the topaction is aborted. The coordinator can also give up on a participant that it is unable to contact, in which case the topaction is aborted. If all participants respond with OK, then the topaction is committed. Once the topaction has aborted or committed, the second phase of the process is entered, where the coordinating AM informs each participant of the outcome, by sending either a `commit` message or an `abort` message. The coordinator continues to send result messages until all participants have acknowledged that they have received and processed a result message.

5.6.2 A Total Order for Topaction Validation

Topaction validating is now a distributed process that occurs at a number of OM's simultaneously. It is necessary to devise a mechanism that ensures that topactions are validated in some consistent global order. At each OM, there is a topaction validation queue, *VQ-TOP*; we must guarantee that topactions enter each OM's *VQ-TOP* in the same relative order. In other words, if topactions *T1* and *T2* have common participants, and *T1* enters a topaction validation queue before *T2* at any common participant, then *T1* must enter before *T2* at *every* common participant. The topaction ordering relation is transitive: if *T1* enters before *T2* at one OM, and *T2* enters before *T3* at another OM, then *T1* must enter before *T3* at all OM's.

The following example shows why a total ordering is necessary. Suppose two topactions, *T1* and *T2*, both read objects x and y, located at object managers *OM-X* and *OM-Y*, and suppose *T1* modifies x and *T2* modifies y. *T1* and *T2* will both send `validate-request` messages to *OM-X* and *OM-Y* when they attempt to commit. Suppose that *T1*'s `validate-request` message arrives at *OM-Y* first, and that *T2*'s message arrives at *OM-X* first. If the topactions simply enter the queues in the order that their `validate-request` messages arrive, then *T1* will enter before *T2* at *OM-Y*, while *T2* will enter before *T1* at *OM-X*. Each OM validates topactions with respect to its own objects. At *OM-X*, the queue has *T2* reading x followed by *T1* reading and writing x; both would validate successfully. At *OM-Y*, the queue has *T1* reading y followed by *T2* reading and modifying y; both would again validate successfully. However, *T1* and *T2* are conflicting actions — there is no equivalent serial order for them. To ensure a correct validation of these topactions, we must ensure that either *T1* validates before *T2* at both *OM-X* and *OM-Y*, or that *T1* validates after *T2* at both OM's.

A global queue-entry order can be achieved using the following approach, which is due to Agrawal, Bernstein, Gupta, and Sengupta [Agrawal *et al.* 1987]. At each OM we keep a new monotonically increasing counter *qet*, the *queue entry time* counter. The idea is to choose the same queue entry time for a topaction at each participating OM. When a topaction T attempts to commit, T's coordinator proposes a *global* queue entry time for the topaction. The proposed entry time is included in `validate-request` messages. At a participant, if the local *qet* counter is less than the proposed queue entry time, then *qet* is set to the proposed time, and topaction T is added to the validation queue. However, if *qet* is greater than or equal to the proposed entry time, *qet* can not be set to this time, since it is monotonically increasing: the participant responds with a special BAD-GLOBAL-TIME response, to inform the coordinator that the proposed time is unacceptable. In this case, the coordinator will choose a later global entry time and send another round of phase-one messages, using `revalidate-request` messages.

This process guarantees a unique global order for entry into OM validation queues. Its only drawback is that choosing a global entry time is potentially a multi-round process. Other possible approaches are discussed in Section 5.7 at the end of the chapter.

5.6.3 Validation at The Coordinator

When topaction T attempts to commit, its AM acts as coordinator of the two-phase commit process. By examining the object identifiers in the read set *rs(T)*, the coordinator can determine the set of participating OM's. During the first phase, each participant is sent a `validate-request` message with the following information: T's action identifier *id(T)*, read set *rs(T)*, write set *ws(T)*, update set *updates(T)*, and a proposed global queue entry time *global-qet(T)*. The update set has pairs of the form `[id(O), shadow-version-value]`, indicating that the shadow version should be installed as the base version of object O. While the updates are not installed until the second phase of the commit, the update set is sent in the `validate-request` message so that OM's can prepare to commit the updates. For example, after responding with an OK response, an OM can assume that the action will commit and start writing out the updates to stable storage. This form of *early prepare* will result in improved processing for commits, which we assume will be more frequent than aborts.

Note that the *entire* read and write sets are sent to each participant, while the update set sent to participant P only contains updates for objects that P manages. It would also be possible to just send partial read and write sets to each participant P; we could send only the read set or write set entries for objects that P manages. However, sending the entire sets allows reordering within a given OM's topaction queue. (Recall that a topaction can move in front of another topaction as long as its read set does not intersect with the write set of the second topaction. It is not possible to determine whether an action can move with only partial read and write sets.)

Figure 5.7 sketches the two-phase commit process managed by topaction T's action manager when T commits. It is only a sketch of the process — it does not cover such occurrences as resending lost messages, aborting because of a timeout, recording the state of the two-phase commit on stable storage, and so on. In addition, it does not show how an appropriate global queue entry time is chosen. We would like to choose a time that is likely to be accepted at all participants; several ways of doing this are discussed Section 5.7. If the coordinator receives a BAD-GLOBAL-TIME response from any participant, it chooses a new global entry time and sends out `revalidate-request` messages; these messages need not include the read set, write set, or update set, since the participants received these in the original `validate-request` messages. (This assumes messages are delivered reliably. If we are using a protocol where messages can be lost or reordered, a participant might receive a `revalidate-request` message without having received a corresponding `validate-request` message. If this were to occur, the participant would ask the coordinator for the missing information.)

5.6.4 Validation at a Participant

The processing of a `validate-request` message by a participant is shown in Figure 5.8. The figure does not cover some of the details of two-phase commit, such as the acknowledgement of a `commit` or `abort` message or the recording of the state of the commit process in stable storage. The processing of a `revalidate-request` message would be similar, except that the topaction's original queue entry would first be removed from the queue.

When a read-write topaction commits, we do not add its entire write set to the participant's *rw-top-committed* set; we only add the part of the write set that is relevant to the participant. This is done because an OM is responsible only for detecting conflicts over its own objects. The function call `OM-copy(ws(H), id(OM-X))` copies only those object identifiers in *ws(H)* that are identifiers of objects managed by participant *OM-X*.

In a distributed pessimistic system, it is usually not necessary to send `commit` or `abort` messages to read-only participants in a topaction, since these participants take exactly the same action (they release the topaction's read locks) regardless of the outcome of the

```
Topaction T Attempts to Commit:
    global-qet(T) := a proposed queue entry time for T
    participants(T) := empty
    for each [id(O), readtime-top] in rs(T) do
        OM-O := O's OM
        if id(OM-O) ∉ participants(T) then add id(OM-O) to participants(T) end
    for each id(OM-P) in participants(T) do
        updates(OM-P) := empty
        end
    for each id(O) in ws(T) do
        OM-O := O's OM
        add the pair [id(O), shadow-version(id(O),T)] to updates(OM-O)
        end
    for each id(OM-P) in participants(T) do
        send the message validate-request(id(T), rs(T), ws(T), updates(OM-P), global-qet(T)) to OM-P
        end
    gather responses...
    if any response is FAILED then
            for each id(OM-P) in participants(T) do
                send the message abort(id(T)) to OM-P
                end
        else if any response is BAD-GLOBAL-TIME then
            global-qet(T) := a new proposed queue entry time for T
            for each id(OM-P) in participants(T) do
                send the message revalidate-request(id(T), global-qet(T)) to OM-P
                end
            go back to gather responses... above
        else if all responses are OK then
            for each id(OM-P) in participants(T) do
                send the message commit(id(T)) to OM-P
                end
        end
```

Figure 5.7: Coordinator, Topaction Validation — Fixed Action Model

```
OM-X receives message:
      validate-request(id(T), rs(T), ws(T), updates(T), global-qet(T))
   begin
      begin critical section
      if global-qet(T) $\leq$ qet then return BAD-GLOBAL-TIME end
      qet := global-qet(T)
      VQcopy := copy(VQ-TOP) ; rw-top-copy := copy(rw-top-committed)
      allocate entry E for action T; E.type := VALIDATING; add E to VQ-TOP
      end critical section
   st(T) := the earliest read time in rs(T)
   for each [ws(RW), et(RW)] pair in rw-top-copy with st(T) < et(RW) do
      for each [id(O), readtime-top] pair in rs(T) do
         if O is managed by this OM and
            id(O) $\in$ ws(RW) and readtime-top < et(RW) then exit FAIL end
         end
      end
   for each read-write action RW in VQcopy do
      for each [id(O), readtime-top] pair in rs(T) do
         if id(O) $\in$ ws(RW) then exit FAIL end
         end
      end
   return value OK
   wait for the message commit(id(T)) or abort(id(T))
   if the message is abort then exit FAIL end
   the message is commit...
      begin critical section
      E.type := READY
      while head(VQ-TOP).type = READY do
         let H be the action at the head of VQ-TOP
         remove H's entry from VQ-TOP
         if H is read-write then
            tnc-top := tnc-top + 1; et(H) := tnc-top
            OM-ws := OM-copy(ws(H), id(OM-X))
            add the pair [OM-ws, et(H)] to rw-top-committed
            for each [id(O), shadow-version-value] in updates(H) do
               base-version(id(O)) := shadow-version-value
               end
            end
         (H is now a committed action.)
         end
      end critical section
   end
      except when FAIL:
               begin critical section
               delete E from VQ-TOP
               end critical section
            return value FAILED
            end
```

Figure 5.8: Participant, Topaction Validation — Fixed Action Model

action. Since read-only participants in this model have no commit or abort processing to do, it would seem that they can simply be removed from the queue after returning an OK response during phase one of the commit, and no second phase message is required. However, this is not the case. An OM might receive and process a `revalidate-request` message at any time, up until the point that it receives a `commit` or `abort` message. (Once it has received a `commit` or `abort` message, it knows that the global entry time was acceptable and the topaction was validated.) Thus, a topaction's entry cannot be removed from an OM's queue until a `commit` or `abort` is received — even if the topaction is read-only.

5.7 Optimizations

This section outlines several optimizations for the fixed action model. In general, our models do not reflect low-level implementation concerns; the optimizations in this section also avoid low-level details. They point out some potential changes that would improve the performance of the model, rather than efficient ways to implement it.

5.7.1 Improving the Global Ordering Protocol

When a coordinator chooses a global entry time for topaction T, we would like it to choose a time that is likely to be greater than all of the *qet* times at T's participants. One way to do this is to have the coordinator poll the participants, asking each for its current *qet* value; the coordinator would choose a global entry time larger than all of these by taking the maximum value and adding some extra amount δ. However, we would like to avoid the extra round of messages that this method requires. One way to do this is to have OM's include their *qet* times in messages that they send to AM's. For example, an OM's response to a `base-version-request` message could include the OM's current *qet* value. An action manager would keep track of the maximum *qet* value it had seen from any OM; it would choose a global queue entry time equal to this maximum plus δ. Conversely, an AM could send OM's the maximum *qet* value it had seen; an OM would set *qet* to the maximum of its current value and the value it received. In this scheme, the maximum values at AM's and the *qet* values at OM's would essentially be a set of loosely synchronized clocks, as described by Lamport [Lamport 1978]. This loose synchronization would mean that most proposed queue entry times would be accepted in one round; there would be conflicts only when two or more actions committed at nearly the same time.

The algorithm as given tests whether a proposed global entry time is later than the entry time of the last action to enter the queue; this is done because we want to add the action to the end of the queue. However, if the proposed entry time would place the action somewhere in the middle of the queue, it is not necessary to reject the entry time if the action can be inserted at the indicated place in the queue. An action can be inserted ahead of other actions in the queue if its write set does not overlap any of their read sets. (Action entry times must be kept in the queue to be able to determine the insertion point for an action entering validation.) A proposed entry time would only be rejected if it was not possible to insert the action in the queue.

Note that some OM's might accept a global queue entry time, and begin validation, while another OM rejects the proposed time. When a `revalidate-request` for a topaction T is received at an OM, T moves back in the queue. If the OM has already started validating T, it does not have to restart validation; it can determine what extra validation must be done, which depends only on the actions that T moved behind.

5.7.2 Another Global Ordering Protocol

Instead of using a set of loosely synchronized clocks to choose queue entry times, it is possible to use a set of tightly synchronized clocks, *i.e.*, a distributed set of clocks where the maximum time difference between any two clocks is less than some known value ϵ (ϵ can be chosen to be a very small number) [Mills 1988]. In such a system, a topaction uses its OM's local clock time as its proposed queue entry time; actions choose entry times that correspond to their real commit order.

Topaction `validate-request` messages will tend to arrive at a given OM with proposed queue entry times properly ordered, since these times correspond to the times that the messages were sent. However, message transit times are not uniform: `validate-request` messages sent at nearly the same time can arrive with proposed entry times that are out of order. At a participant, topaction T can enter the validation queue as soon as it arrives, as long as its proposed entry time *global-qet(T)* is not earlier than the entry time of the last action to enter the queue. However, it makes sense to wait until local time *global-qet(T)* $+ \delta$ before allowing T to enter the queue, where the extra amount δ is an attempt to compensate for possible reorderings due to different message transit times. While waiting for time *global-qet(T)* $+ \delta$, if another `validate-request` message arrives with a proposed time that is earlier than *global-qet(T)*, then this action is added to the queue before T.

If a `validate-request` message with proposed time *global-qet(T)* arrives before local time *global-qet(T)* + δ, it will always be allowed into the queue, even if some earlier message with an out-of-order entry time has already arrived. Thus, a good choice for δ would be a time slightly larger than the expected message transit time. Even if a message for topaction T arrives after local time *global-qet(T)* + δ, T can still enter the queue as long as its proposed entry time is greater than the entry time of the latest action to enter the queue; a BAD-GLOBAL-TIME response is only used if this is not the case. (Note that δ can be adjusted dynamically according to the frequency of BAD-GLOBAL-TIME responses.)

5.7.3 Caching

Compared to reading a local shadow version of an object, reading a base version over the network is extremely expensive. For this reason, the most important performance gains for this model will result from reductions in the number of `base-version-request` messages that are sent. One obvious optimization that has this effect is object caching.

As the model is described above, a `base-version-request` message is sent every time a base version is read — if an action reads an object 20 times, 20 `base-version-request` messages will be sent (assuming no shadow versions exist, *i.e.*, the object has not been modified). We would like to cache base versions to avoid this kind of redundancy; at most one message should be sent per object read by an action. We would also like to cache objects *across* topactions, so that if one topaction reads a base version, other topactions at the same AM can read the cached version afterwards. Thus, each AM should have a single base version cache that is updated each time a base version is requested from an OM. A `base-version-request` message will only be sent if this cache does not have a base version for the object being read. When a base version is stored in an AM's cache, its *OM-readtime* is stored along with it. This read time is recorded in read set entries when a cached base version is read. Thus, read set entries still record the time that the base version was read at its OM.

Introducing a cache at each AM introduces a cache consistency problem: after an object O is updated at its object manager, caches can have out-of-date base versions for O, and actions can read these out-of-date values. Since an action that reads an out-of-date value will fail validation, this consistency problem does not affect the correctness of the system. (As we pointed out in the introduction, this is an advantage over pessimistic systems, where reading an out-of-date value is not allowed.) However, the caches do need to be updated eventually.

There are two general approaches to cache updating: *lazy* and *eager*. In a lazy approach, we wait until an action fails validation to update out-of-date caches. If an OM validating topaction T find that T had read out-of-date base versions, it includes the new values for these base versions in the FAILED response that it returns to T's AM; these new values are installed in the AM's cache.

In an eager approach, OM's keep track of which AM's are caching which values. When an object O is updated, its OM sends update messages to all AM's that are caching O. Although this approach attempts to inform AM's of out-of-date values as early as possible, there is no constraint on when these update messages arrive: validation will catch all out-of-date reads. Thus, update messages can be grouped together and sent as a single message, or "piggy-backed" on other messages sent to the AM's, such as `base-version-request` responses.

Eager updating has the advantage that it may prevent some actions from aborting due to a read of an out-of-date cache. Lazy updating has the advantage that less update information is sent.

5.7.4 Grouping Base Versions

Another optimization that reduces the number of `base-version-request` messages stems from the following observation: when a base version of an object is retrieved from an OM, if the value includes object identifiers of objects then the action reading the object may later read some of these referenced objects. If any of these other objects are managed by the OM responding to the `base-version-request`, then we can send copies of their base versions along with the copy of the base version that was requested. All base versions returned from a `base-version-request` would be installed in the action manager's cache. Each OM would keep track of the objects it has sent to an AM, so that it would not resend objects that it had already sent. The number of related objects sent would depend on parameters such as the size of the objects referenced, the maximum size of a network packet, statistics of object usage, and so on. Note that returning additional base versions does not affect the probability of committing. An action still sends its read set to OM's for validation; objects that were sent but not read will not lead to additional conflicts.

Systems could support explicit grouping of base versions by programmers or a database administrator. Rather than traversing an object's references dynamically, the object's group is used: when a base version from a given group is requested, copies of all base versions in the group are returned.

5.7.5 Action Manager Topaction Validation

If two conflicting topactions run at the same AM, this can be detected by the AM *before* it acts as the coordinator of a distributed commit. When a topaction attempts to commit at an AM, the AM first validates the topaction against all committed local topactions; to do this, an AM keeps its own *rw-top-committed* set. An AM resorts to the full distributed commit process only if the topaction passes the local topaction validation process.

Chapter 6

Fixed Object Model

The single-site model described in Chapter 4 has the following constraints:

1. A subaction can only run at the same entity as its parent action.

2. An action can only read or write objects managed by the entity where it is running.

The first constraint simplifies the validation of subactions, since all sibling subactions must run at the same entity; the second constraint simplifies the management of base versions, since the `nearest-version` function will always be able to access base versions directly.

If we eliminate either of these constraints, the result is a distributed transaction system. In the last chapter, the fixed action model eliminates the second constraint: subactions are required to run at the same action manager as their parent, but an action can read any object in the system. Subaction validation remains as simple as in the single-site model, but base version management becomes more complicated. Reading a base version involves sending a `base-version-request` message; installing shadow versions at topaction commit time involves sending update sets to object managers.

In this chapter, we present the *fixed object model*, which eliminates the first constraint but keeps the second: a subaction does *not* need to run at the same entity as its parent, but an action can only read or write objects managed by the entity where it is running. In this model, managing base versions remains as simple as in the single-site model, but subaction validation becomes complicated.

In this model we only have one type of entity, the object manager. Since an action can only read or write an object located at the entity where it is running, actions must run at OM's; we do not have AM's. Object managers manage both subactions and topactions, as well as performing base-version management. Shadow versions are created when an action first writes an object: in this model all shadow versions are located at the OM of their base version.

In the fixed action model the objects move to the actions; in this model the actions move to the objects. If an action at one OM wants to access some objects located at another OM, the action starts a subaction at the other OM that will perform the desired reads and writes.

The models are equivalent in terms of computational power; however, more actions may be necessary in this model to perform the same computation. The following example illustrates the differences between the two models. Suppose objects A and B are located at distinct object managers *OM-A* and *OM-B*, respectively, and the computation we wish to perform is to read object A, read object B, and then write object B.

- In the fixed action model, a single topaction is required, executing at any AM in the system.

  One `base-version-request` is used to read A, and another is used to read B. A third `base-version-request` is used when writing B; the copy of B that is returned is installed as a shadow version of the topaction, and this shadow version is modified. When the topaction commits, the shadow version is sent to *OM-B* in an update set, and is installed as B's base version.

- In the fixed object model, two actions are required to perform the computation: a topaction at *OM-A* and a subaction at *OM-B*.

  The topaction at *OM-A* reads the base version of A. It then starts a subaction at *OM-B*, which reads the base version of B. To write B, the subaction creates a shadow version of B which it then updates. When the topaction commits (at *OM-A*), it informs *OM-B* that it has committed; *OM-B* then installs the shadow version of B as a new base version. The shadow version is *not* sent in a message, since it is already located at *OM-B*.

Figure 6.1 gives an example of a set of object managers in the fixed object model. Topaction A has started two subactions, *A.1* and *A.2*; *A.1* is running at A's OM, *OM-X*, while *A.2* is running at *OM-Y*. The figure shows the shadow versions for each action; these versions are always shadows of local base versions.

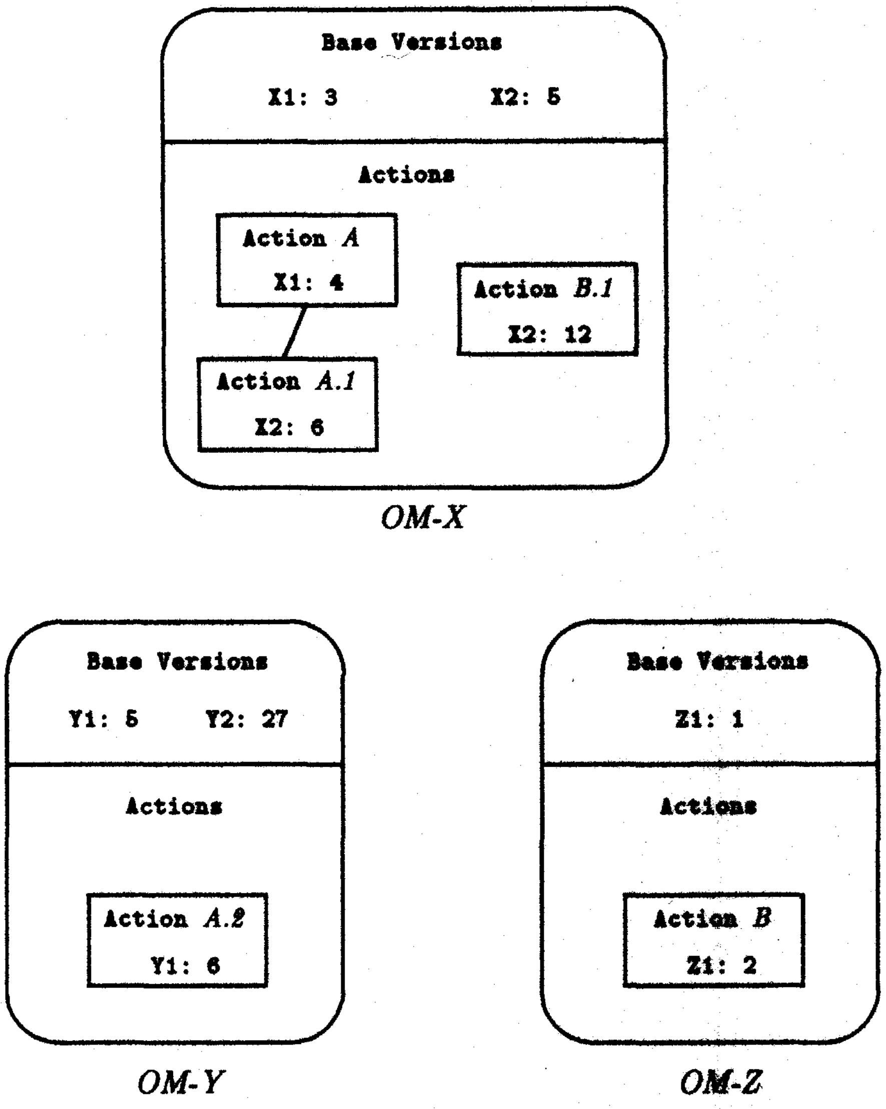


Figure 6.1: Object Managers — Fixed Object Model

6.1 Overview

Because this model is more complicated than any of the previous models, we give a brief overview before describing it in detail.

In this model, subaction management involves communication between the OM's in the system. Three message types are used: `start-subaction` messages, which we use to start a subaction; a `commit-request` messages, which we use to commit a subaction; and `subaction-commit` messages, which we use to inform OM's about a subaction commit.

When an action at one OM wishes to start a subaction at another OM, it sends a `start-subaction` message to the other OM. Since a subaction need not run at the same OM as its parent, sibling actions might run at different OM's. The obvious place to validate a set of siblings is at the OM where their parent is running. When a subaction attempts to commit, its OM sends a `commit-request` message to the OM of its parent. Thus, a message is sent to start a subaction, and a return message is sent to commit the subaction.

In the fixed action model, a committed subaction's shadow versions, which are all located at the same AM, become shadow versions of its parent action in one local step. In the fixed object model, a subaction's shadow versions are not necessarily all located at the same entity; an action's descendants may have modified objects at any number of OM's. When a subaction C attempts to commit to parent P, P's OM validates C; if validation succeeds, P's OM sends a `subaction-commit` message to each OM that is a *read-write participant* in C. An object manager *OM-X* is a read-write participant in C if C or any descendant that committed up to C ran at *OM-X* and modified one of *OM-X*'s objects. When a read-write participant receives a `subaction-commit` message for C, it moves the appropriate shadow versions up to parent P. Read-only participants do not have to be informed of a subaction commit, since they do not have any shadow versions that need to be moved.

Unfortunately, the approach of validating a subaction at its parent's OM results in a complete horizontal partitioning of the actions in the system; *i.e.*, the actions in the system are partitioned into the following sets: the topactions, the children of topactions, the children of the children of topactions, and so on. If every ancestor of an action is located at a different OM, then each validation of a subaction's read set (when the subaction commits, when its parent commits, when the parent of its parent commits, *etc.*) will take place at a different OM. For a complete horizontal partitioning, a subaction at level k in an action tree (where a topaction is at level one) must record a k-part timestamp for read times.

Topaction validation turns out to be almost identical to topaction validation in the fixed action model. When a topaction attempts to commit, its OM acts as the coordinator of a two-phase commit, just as a topaction's AM acted as coordinator in the fixed action model. Every OM that ran a descendant of the topaction is a participant in the topaction, and is asked by the coordinator to validate it. The only difference between an AM coordinator for the fixed action model and an OM coordinator for this one is that an OM coordinator does not send update sets to participants, since the shadow versions are already located at the participants.

6.2 Data Structures at an Object Manager

Subaction management and validation is considerably more complicated here; we need a number of data structures that were not needed in either the single-site model or the fixed action model. The most important change is the need for multi-part timestamps, or *multistamps.*

6.2.1 Multistamps for Subaction Read Times

Each OM has a data structure *latest*, which keeps track of the latest `subaction-commit` it has received from each other OM. When object manager *OM-P* of a parent action P validates and commits a child C, it sends out `subaction-commit` messages to each read-write participant; these messages include the end time *et(C)* of the committed subaction. When read-write participant *OM-X* receives the `subaction-commit` message from *OM-P*, it moves the appropriate shadow versions up to P and sets the value of *latest(OM-P)* to *et(C)*. Object managers must use a message protocol that ensures that `subaction-commit` messages are applied in the same order that they are sent.[1] Thus, if the value *latest(OM-P)* is 27 at object manager *OM-X*, then *OM-X* has processed all `subaction-commit` messages sent from *OM-P* with end times less than or equal to 27. The *latest* data structure gives a compact summary of what `subaction-commit` messages have been processed by an OM.

The *latest* data structure is used for recording the multi-part timestamps that are needed for subaction read times. Consider a subaction C running at *OM-C*, where C has parent B running at *OM-B*, and B has topaction parent A running at *OM-A*. When C reads an object, what must be recorded in the three-part timestamp? When C attempts to commit, object manager *OM-B* will validate the read time; when B attempts to commit, object

[1] More precisely, the `subaction-commit` messages sent from one OM to another must be processed by the receiving OM in the same order that they were sent by the sending OM, with no "gaps" in the sequence.

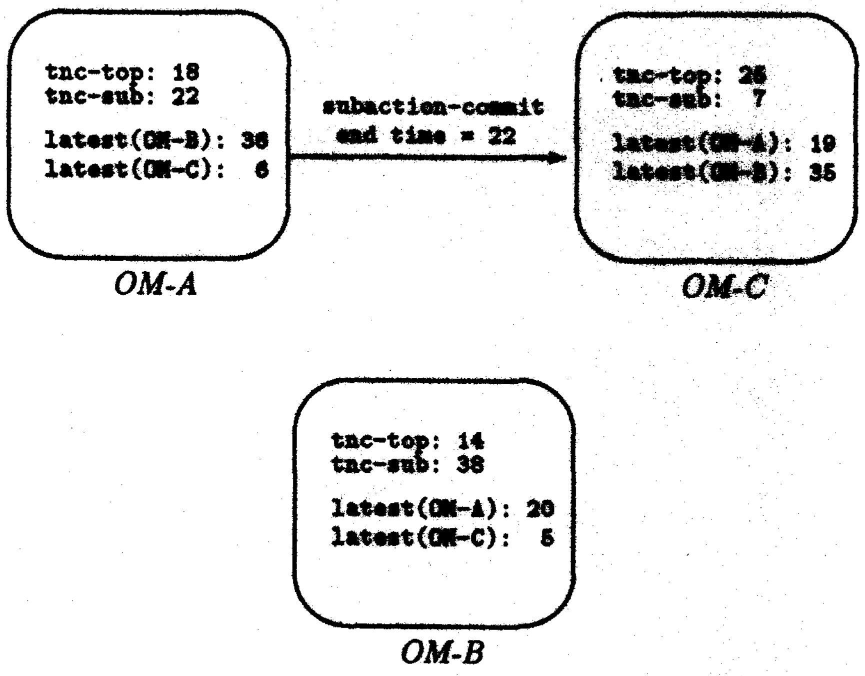
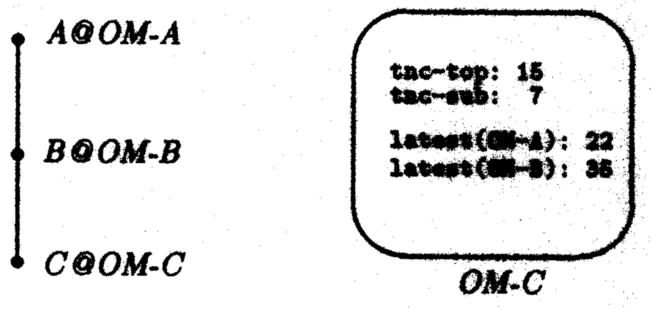


Figure 6.2: The *latest* Data Structure — Fixed Object Model

(a) Action tree for topaction *A*; current *OM-C* data



(b) Action *C* reads object *O*

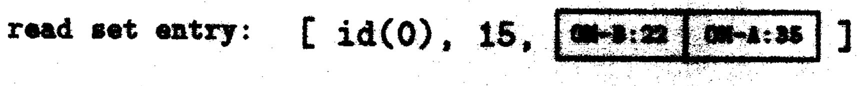


Figure 6.3: Multistamp Example — Fixed Object Model

manager *OM-A* will validate the read time; and when topaction A commits, object manager *OM-C* will validate the read time (as a participant in the two-phase commit process). For topaction validation, *OM-C*'s *tnc-top* value, the transaction counter for *OM-C*'s topaction validation queue, is recorded. Since the read is occurring at *OM-C*, this is a local value. For subaction validation, the values *latest(OM-B)* and *latest(A)* are recorded. Consider *latest(OM-B)*, which is the end time of the latest subaction commit that occurred at *OM-B* to have been processed at *OM-C*. This end time was a *tnc-sub* value from *OM-B*, and is precisely what *OM-B* will need to know when it validates the read that occurred at *OM-C*.

At commit time, we record a single *tnc-top* value and a number of *latest(OM-X)* values. For convenience, we separate the two: a read set entry for a subaction read set has the form `[id(O), multistamp, tnc-top-value]`.

Multistamps are used for subaction validation: they only record *latest(OM-X)* values. For a subaction A, we define `take-multistamp(A)` to mean: create and return a multistamp that contains the *latest(OM-X)* value of each object manager *OM-X* that is running a proper ancestor of A. If M is a multistamp, then `M[OM-X]` extracts the *latest(OM-X)* value that was recorded for *OM-X*. The function call `truncate(M, P)` returns a new multistamp that is a copy of M that does not include *latest(OM-X)* if *OM-X* is not running a proper ancestor of P. When a child C commits to parent P, each k-part multistamp M in C's read set can be converted to a $(k-1)$-part multistamp using `truncate(M, P)`. Finally, if *M1* and *M2* are multistamps, then the function call `merge(M1, M2)` returns a new multistamp that has values for all of the OM's in either *M1* or *M2*. If both *M1* and *M2* have a *latest* value for *OM-X*, then the *earlier* *OM-X* value is retained. When child C commits to parent P, if both C and P have read an object O, then `merge` can be used to merge C's and P's multistamps for O.

Figure 6.2 shows a system with three object managers, *OM-A*, *OM-B*, and *OM-C*. Each OM has two *latest* values, one for each of the other OM's. At object manager *OM-A*, *tnc-sub* is 22; 22 read-write subactions have committed at *OM-A*. Examining the *latest(OM-A)* values at the other OM's, we can see that *OM-B* has processed all *OM-A* commits through commit 20, while *OM-C* has processed all *OM-A* commits through commit 19. The most recent commit, at time 22, caused a `subaction-commit` message to be sent to *OM-C*, as is shown in the figure; *OM-C* must be a read-write participant in the action that committed. The `subaction-commit` message includes the end time of the committing action, 22, and *OM-C* will change its *latest(OM-A)* value from 19 to 22 when it processes this message.

Part (a) of Figure 6.3 shows an action tree for the same system: subaction C at *OM-C* has parent B at OM-B, and B has parent A at *OM-A*; it also shows *OM-C*'s current state. Part (b) shows the read set entry that is recorded at *OM-C* when action C reads object O. The tripple consists of O's identifier, the current tnc-top value, and a multistamp that records *latest* values for all OM's running proper ancestors of action C. In this case, the current tnc-top value is 25, and the multistamp records the values *latest(OM-B)* and *latest(OM-A)*. If this multistamp is M, then `M[OM-A]` is 22 and `M[OM-B]` is 35.

6.2.2 The Remaining Data Structures

In addition to the *latest* data structure, each OM has the following data structures. For topaction validation, each OM has a validation queue *VQ-TOP* and associated counter *tnc-top*. For global topaction ordering, each OM has a queue entry time counter, *qet*. For subaction validation, each OM has a validation queue *VQ-SUB* and associated counter *tnc-sub*. At each OM, *base-version(id(O))* gives the base version of object O, and *shadow(A, id(O))* gives action A's shadow version of object O, if it exists.

Each OM also has a *status* data structure. For any action A, *status(A)* indicates the status of the action, one of the following: ACTIVE, COMMITTED, or ABORTED. By default, an action's status is ACTIVE; its status is explicitly set to COMMITTED (or ABORTED) when its shadow versions are moved (or discarded).

For every action A that runs at an OM, the OM keeps some information about A and its children. The information about A's children is used to validate them when they attempt to commit: A's OM keeps the set *rw-committed(A)*, which contains information about the read-write committed children of A. The information about A is a read set *rs(A)*, a write set *ws(A)*, an aborted set *aborted(A)*, and a participants set *participants(A)*. When A attempts to commit, these sets are sent to the OM of A's parent.

A topaction's read set contains pairs of the form `[id(O), tnc-top]`; a subaction's read set contains triples of the form `[id(O), multistamp, tnc-top]`. The write set for any action contains object identifiers, as in the other models.

For any action A, *aborted(A)* contains the action identifiers of aborted descendants of A; *participants(A)* keeps identifies the OM's that ran A or one of its descendants. Each entry in *participants(A)* is a pair `[id(OM-X), read-write]`: *id(OM-X)* is the identifier of participant *OM-X*, and *read-write* is a boolean indicating whether A (or a descendant that has committed up to A) has modified any objects at *OM-X*. If the boolean is true, the object

```
Function nearest-version(A, id(O)):
    if shadow(A, id(O)) exists then return shadow(A, id(O)) end
    if A is a topaction then
            return base-version(id(O))
        else
            return nearest-version(parent-of(A), id(O))
        end
```

Figure 6.4: The `nearest-version` Function — Fixed Object Model

manager *OM-X* is a *read-write participant* in *A*, otherwise it is a *read-only participant.*

Note the differences from the fixed action model. In the fixed object model, information about an action's committed read-write siblings is kept at the OM of its parent. In addition, there are two new sets for each action, *aborted(A)* and *participants(A)*, and there are two new data structures at each OM, *latest* and *status*. Each OM manages base versions *and* shadow versions.

Figure 6.4 gives the `nearest-version` function for this model. Base versions are accessed directly, as was done in the single-site model: Figure 6.4 is identical to Figure 4.2.

6.3 Read and Write

The rules for reading and writing an object are given in Figure 6.5. They are very similar to those for the single-site model (Figure 4.3); the only difference is the use of multistamps for subaction read times. When an object is read, both topactions and subactions record the current *tnc-top* value, which is used for topaction validation; a subaction *A* also uses `take-multistamp(A)` to record a multistamp to be used for subaction validation.

6.4 Starting a Subaction

A subaction can be started at the same OM as its parent action, or at another OM; we call the first kind of subaction a *local subaction*, and the second a *remote subaction.* In general, we will only discuss remote subactions; *i.e.*, we assume that starting a subaction or committing a subaction involves sending a message from one OM to another. The discussion nevertheless applies to local subactions, with two differences. First, `start-subaction` and `commit-request` messages are not required for a local subaction. Second, the failure modes for the subaction and its parent are not independent. Messages are not lost, and a

```
Action A reads Object O:
    if id(O) ∉ rs(A) then
        if A is a topaction then
                add the pair [id(O), tnc-top] to rs(A)
            else
                add the triple [id(O), take-multistamp(A), tnc-top] to rs(A)
            end
        end
    read nearest-version(A, id(O))

Action A writes Object O:
    if id(O) ∉ ws(A) then
        add id(O) to ws(A)
        shadow(A, id(O)) := copy(nearest-version(A, id(O)))
        end
    modify shadow(A, id(O))
```

Figure 6.5: Read and Write — Fixed Object Model

subaction's OM cannot fail while its parent's OM does not; any discussion of message or OM failure does not apply to local subactions.

When an action P at *OM-P* wants to start a remote action C at *OM-C*, it creates an action identifier for C, *id(C)*, and sends the message `start-subaction(id(C))` to *OM-C*. An action's identifier includes the identifier of the OM where it is running, as well as the action identifiers of all parent actions. Thus, we can determine from *id(C)* that *OM-P* is the OM of C's parent action. Action P then blocks until action C sends a `commit-request` message to *OM-P*. At *OM-C*, when the `start-subaction` message is received, C is started and run; when C attempts to commit, a `commit-request` is sent to *OM-P*. *OM-P* validates C when the `commit-request` message arrives; P proceeds with its own execution after C has committed or aborted.

If an action wishes to start a number of concurrent children, it sends a set of `start-subaction` messages and then blocks; each child is validated when it sends the parent's OM a `commit-request`, and the parent only continues its own execution after all of its active children have committed or aborted. In some cases, however, the parent may wish to continue before some of the concurrent children have sent `commit-request` messages. For example, a parent that starts two equivalent children might want to continue after either of the children has committed. A parent can unilaterally abort any of its active children by adding them to its aborted set. Thus, if a parent wanted to continue before some of its children had finished, it would abort all remaining active children and then continue.

Because there can be communications failures or node crashes, a response that an action is waiting for may never arrive. For example, a `commit-request` from a subaction may never arrive. After waiting a reasonable period of time for a `commit-request` (and possibly after sending query messages to find out if a subaction is still running) the OM of the parent action can decide that there has been a failure, in which case it can unilaterally abort the child.

6.5 Subaction Commit Process

The subaction commit process involves a number of OM's: the subaction's OM sends a `commit-request`; the subaction's parent's OM validates the subaction; and, if validation succeeds, the parent's OM sends `subaction-commit` messages to the read-write participants in the subaction. The next three sections cover these three processes.

6.5.1 Processing at a Subaction's OM

The process of sending a subaction `commit-request` message is shown in Figure 6.6. When a child C running at *OM-C* wants to commit to its parent P running at *OM-P*, it first it adds its own OM to its participants set: it adds the pair `[id(OM-C), read-write]`, where read-write is a boolean indicating whether C or any of its descendants has modified an object managed by C. This boolean is determined by examining C's write set. Note that C may have a descendant that ran at *OM-C* and committed up to C; in this case, there will already be an entry for *OM-C* in C's participants set, which we remove before adding the new one. The *read-write* boolean is still calculated correctly by examining C's write set, since the write set includes the writes of committed descendants. Next, the subaction sends a `commit-request` to the OM of its parent. This message includes the subaction's identifier, read set, write set, participants set, and aborted set.

From the point of view of the subaction, subaction commit process is now finished. If validation succeeds at the parent, then the subaction's OM may be one of the read-write participants that is informed of its commit. However, it is treated like any other read-write participant.

```
Subaction C at OM-C with Parent P at OM-P Attempts to Commit:
    if there is pair [id(OM-C), boolean] in participants(C) then remove it end
    if there is an id(O) ∈ ws(C) such that O is managed by OM-C then
            read-write := true
        else
            read-write := false
        end
    add the pair [id(OM-C), read-write] to participants(C)
    send the message commit-request(id(C), rs(C), ws(C), participants(C), aborted(C))
            to OM-P
```

Figure 6.6: Sending a `commit-request` Message — Fixed Object Model

6.5.2 Processing at a Subaction's Parent's OM

When the `commit-request` is received at the parent's OM, the subaction is validated against its committed read-write siblings. Information about these siblings is available, since the siblings must have been validated by the OM of the parent. If validation fails, the parent's OM simply adds the subaction's identifier to its aborted set. If validation succeeds, the information about the child (read set, write set, participants set, aborted set) is merged with its parent's information. The parent's OM then sends `subaction-commit` messages to all read-write participants in the action. The purpose of these messages is to move the appropriate shadow versions up to the level of the parent. Messages are not sent to read-only participants, since these have no shadow versions to be moved. Similarly, no messages are sent if the subaction is aborted, since an abort does not result in any shadow version movement. A `subaction-commit` message includes the committed subaction's end time and aborted set; the end time is used for setting the value *latest(OM-P)* at each read-write participant, and the aborted set is used to make sure that the shadow versions of aborted actions are not moved up to the parent.

Figure 6.7 shows the validation process for object manager *OM-P* when it receives a `commit-request` message. *OM-P*'s subaction validation queue, *VQ-SUB*, is used for validation. *OM-P* extracts its own value, *M[OM-P]*, from each read-time multistamp *M*. This value, which was recorded at *OM-R*, the OM where the read occurred, is the end time from the last `subaction-commit` message from *OM-P* to have been processed by *OM-R* when the read occurred. The value is compared to the end times of committed siblings that has written the object; if any sibling's end time is greater than the end time extracted from the multistamp, then the committing subaction must have read an out-of-date value, and validation fails.

At *OM-P*, action C is committed to P by merging its various sets (read set, write set, aborted set, participants set) into P's sets. Figure 6.8 shows this merging process. When merging read sets, the subaction's multistamp is truncated from a k-part multistamp to a $(k-1)$-part multistamp using the `truncate` operation. If the subaction and its parent both have a multistamp read time for the same object, then the `merge` operation is used to produce a new multistamp. (Section 6.2.1 describes the `truncate` and `merge` operations for multistamps.) When merging participants sets, if the subaction and its parent both have an entry for the same OM, then the boolean-or of their two *read-write* booleans is used; *i.e.*, if either entry indicates that the participant is read-write, then the participant is read-write in the merged set.

6.5.3 Processing at a Read-Write Participant

As `subaction-commit` messages arrive at a read-write participant, they are processed in the same order that they were sent. In other words, if an object manager sends `subaction-commit` messages *M1* and *M2* to *OM-X* (in that order), then *OM-X* first processes *M1*, and then processes *M2*.

The processing of a `subaction-commit` message that commits child C up to parent P is shown in Figure 6.9. First, *C's* shadow versions are moved up to P. The value *latest(OM-P)* is then set to the end time that was sent in the `subaction-commit` message; this indicates that all `subaction-commit` messages from *OM-P* with end times less than or equal to that end time have been processed. This value is only updated after the shadow versions have been moved; read times must not record the new *latest(OM-P)* time prior to this point, since actions can continue to run while `subaction-commit` messages are processed. However, an action can read a new value but record an old *latest(OM-P)* value, which would lead to an unnecessary abort. Actions could block during `subaction-commit` processing, in which case this would not occur. Whether actions should block depends on how the cost of blocking compares to the cost of restarting those actions that abort because they record an older *latest(OM-P)* value.

Shadow versions movement occurs as follows. First, the set *aborted(C)* is used to change the status of each aborted action and its descendants to ABORTED. Actions are marked as ABORTED so that their shadow versions will not be moved during the next step. The function call `descendants(A)` gives all descendants of A about which the OM has information: some

```
The message commit-request(id(C), rs(C), ws(C), aborted(C), participants(C))
        is received at OM-P (C's Parent is P):
  begin
    begin critical section for subactions
    VQcopy := rw-sibling-copy(VQ-SUB, C) ; rw-copy := copy(rw-committed(P))
    allocate entry E for action C
    E.type := VALIDATING; add E to VQ-SUB
    end critical section for subactions
  st(C) := the earliest OM-C read time M[OM-C] from any multistamp M in rs(C)
  for each [ws(RW), et(RW)] pair in rw-copy with $st(C) < et(RW)$ do
    for each [readobj, M, readtime-top] triple in rs(C) do
      if readobj $\in$ ws(RW) and $M[OM-P] < et(RW)$ then exit FAIL end
      end
    end
  for each read-write action RW in VQcopy do
    for each [readobj, M, readtime-top] triple in rs(C) do
      if readobj $\in$ ws(RW) then exit FAIL end
      end
    end
    begin critical section for subactions
    E.type := READY
    while head(VQ-SUB).type = READY do
      let H be the action at the head of VQ-SUB
      remove H's entry from VQ-SUB
      if H is read-write then
        tnc-sub := tnc-sub + 1; et(H) := tnc-sub
        add the pair [ws(H), et(H)] to rw-committed(P)
        for each [id(OM-X), read-write] in participants(H) do
          if read-write then
            send the message subaction-commit(id(H), aborted(H), et(H)) to OM-X
            end
          end
        end
      commit action H
      (H is now a committed action.)
      end
    end critical section for subactions
  end
    except when FAIL:
          begin critical section for subactions
          delete E from VQ-SUB
          add id(C) to aborted(P)
          end critical section for subactions
        signal FAILED
        end
```

Figure 6.7: Subaction Validation — Fixed Object Model

```
Commit Action C (to Parent P):
    if P is a topaction then
        for each [id(O), M, C-rt-top] in rs(C) do
            if there is an [id(O), P-rt-top] in rs(P) then
                    replace [id(O), P-rt-top] with
                            [id(O), min(C-rt-top, P-rt-top)] in rs(P)
                else
                    add [id(O), C-rt-top] to rs(P)
                end
            end
        end
    if P is a subaction then
        for each [id(O), C-M, C-rt-top] in rs(C) do
        new-C-M := truncate(C-M, P)
            if there is an [id(O), P-M, P-rt-top] in rs(P) then
                    replace [id(O), P-M, P-rt-top] with
                            [id(O), merge(new-C-M, P-M), min(C-rt-top, P-rt-top)] in rs(P)
                else
                    add [id(O), new-C-M, C-rt-top] to rs(P)
                end
            end
        end
    for each id(O) in ws(C) do
        if id(O) is not in ws(P) then add id(O) to ws(P) end
        end
    for each id(A) in aborted(C) do add id(A) to aborted(P) end
    for each [id(OM-X), boolean-C] in participants(C) do
        if there is an [id(OM-X), boolean-P] in participants(P) then
                replace [id(OM-X), boolean-P] with
                        [id(OM-X), (boolean-C or boolean-P)] in participants(P)
            else
                add [id(OM-X), boolean-C] to participants(P)
            end
        end
```

Figure 6.8: Subaction Commit-Time Update — Fixed Object Model

```
The message subaction-commit(id(C), aborted(C), et(C))
        is received at OM-X (C's Parent is P):
    for each id(A) in aborted(C) do
        for each D in descendants(A) do
            abort D if it is still running
            status(D) := aborted
            end
        end
    for each A in descendants(C) do
        if status(A) = active then
            for each existing shadow(A, id(O)) do
                shadow(P, id(O)) := shadow(A, id(O))
                end
            status(A) := committed
            end
        end
    latest(OM-P) := et(C)
```

Figure 6.9: Processing a `subaction-commit` Message — Fixed Object Model

appropriate data structure should be kept so that this set can be computed quickly for any action A. Note that the descendants of A includes A itself. Some of the actions that have their status changed to ABORTED may still be running. For example, if an action starts a subaction, but never receives a `commit-request` from that subaction, it might abort, while its child action continues to run. This child action is called an *orphan*. If an action whose status is being changed to aborted is still running, then it is aborted: it is forced to stop running, and its data structures are discarded. Next, the shadow versions of C and any descendant of C that still has status `active` are moved up to P.

If all `subaction-commit` messages for C's descendants have already been processed, then all shadow versions for actions that have committed up to C have already been moved up to C: only C itself will have shadow versions to be moved up to P. However, it is possible that some `subaction-commit` messages for C's descendants have not been processed yet; `subaction-commit` messages can arrive out of order, if they originate from different OM's. For example, an action might commit at one OM, causing `subaction-commit` messages to be sent for it, and its parent might commit at another OM, causing `subaction-commit` messages to be sent for the parent. Since two different OM's send these messages, there is no guarantee that the parent's `subaction-commit` message will arrive second at a given read-write participant. This potential reordering does not matter: our shadow version update

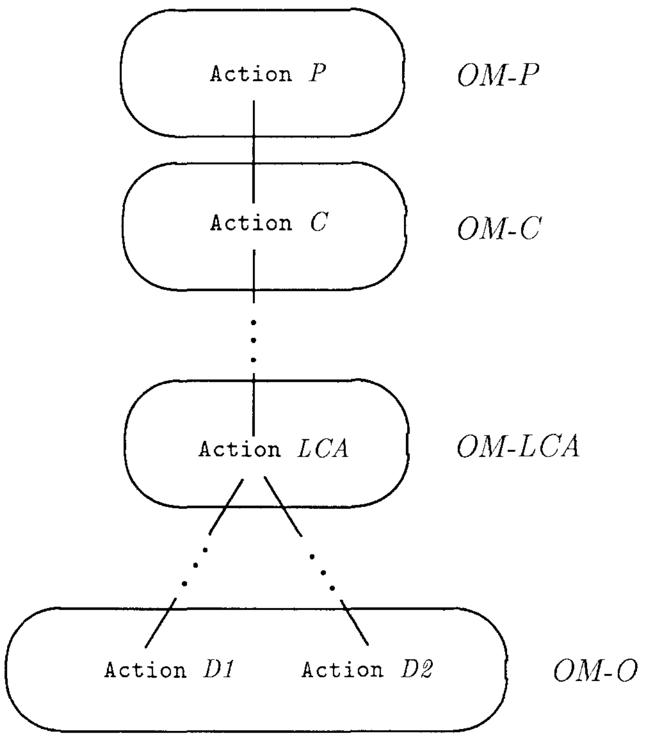


Figure 6.10: Subaction Commit Example — Fixed Object Model

algorithm works correctly regardless of the `subaction-commit` arrival order.

Our algorithm is *idempotent.* Committed action remain committed, aborted actions remain aborted, active actions become committed or aborted only once, and shadow versions are always moved up to an active action. Thus, regardless of order, actions will be given the correct status, and only the shadow versions of committed actions will be moved up to an active action. (It will never be the case that the shadow version of an aborted action is moved up to an active action.)

While idempotence guarantees some properties, there is one other concern. Suppose two descendants of a committing subaction *C*, *D1* and *D2*, have both written an object *O* at object manager *OM-O*, but a `subaction-commit` message has not been received by *OM-O* for either *D1* or *D2*. This situation is shown in Figure 6.10. When the `subaction-commit` for action *C* arrives, if neither *D1* nor *D2* is a descendant of an aborted action, then both *D1* and *D2* have a version of *O* to be moved up to *P* — it is not clear which version should become *P*'s new shadow version.

It turns out that this scenario can never occur; it leads to a contradiction, as follows. Both *D1* and *D2* must have run at *O*'s OM, *OM-O*. Note that *D1* can not be an ancestor of *D2*, or *vice versa*: an ancestor does not commit until its descendants have committed or aborted, and neither *D1* nor *D2* is known (at *OM-O*) to have committed or aborted. *D1* and *D2* must have committed up to their least common ancestor, *LCA* (which will either be committing action *C* or some descendant of *C* that has committed up to C). *D1* must have read *D2*'s version of *O*, or *vice versa*; otherwise, a conflict would have been detected when the updates of *O* were committed to *LCA*'s object manager. Neither *D1* nor *D2* is a descendant of an aborted action; thus, a conflict was not detected. Since one has read the other's version, at the time of that read it must have been the case that one or the other was *already* committed up to *LCA* at *OM-O* — but this contradicts our premise that neither has status committed at *OM-O* when the `subaction-commit` message for *C* arrives.

This shows that for any `subaction-commit` from a child *C* to a parent *P*, no two shadow versions moved up to *P* will be for the same object *O*. This result holds even when `subaction-commit` messages arrive out of order or are lost.

6.5.4 Discussion

It is worthwhile to step back and examine the subaction commit process we have just described. In the fixed action model, all processing of a subaction commit occurs locally. The action is validated locally, and, if it commits, its shadow versions are moved up to its parent locally, at the same time. In the fixed object model, an action is validated by the OM of its parent, and, if it commits, a set of `subaction-commit` messages is sent to all OM's that are read-write participants in the action, which then move the action's shadow versions. Some of these messages may never arrive, and they certainly do not all arrive at the same time.

Thus, at any time after a subaction commits, the commit may or may not have been processed by one of the OM's where it ran. Because of this, we must record at each OM information about which subaction commits have been processed. The *latest* data structure is used for this purpose. If object manager *OM-A* has a value of 27 for *latest(OM-X)*, then the `subaction-commit` for every subaction that committed at object manager *OM-X* with an end time of 27 or earlier has been processed by *OM-A*. To maintain this data structure, it is necessary for an OM to process `subaction-commit` messages sent from a given OM in the order of their end times — messages between each *pair* of OM's must be ordered.

The *latest* data structure is used to record subaction read times. A read by subaction will potentially be validated by all of its ancestors' OM's. At a given object manager *OM-X*, validation is concerned with actions that committed at *OM-X*. To validate the reading of some object O, *OM-X* must know which actions that committed at *OM-X* had been processed by O's OM at the time that O was read; *i.e.*, it must know what the value of *latest(OM-X)* was at the time of the read. Thus, when an action A reads an object, a multistamp read time is recorded that includes the *latest* value for each OM that is running a proper ancestor of A.

No OM has to block after sending a message while another OM processes the message. When a subaction attempts to commit, its OM sends a `commit-request` message, and can then continue executing other activities. If the parent's OM commits the subaction, it sends out `subaction-commit` messages and forgets about them. As we showed, it does not matter if `subaction-commit` messages are lost; the commit of an action can always be processed, even if the commits for some of its descendants were lost or have not arrived yet. In the worst case, the arrival of a topaction commit message will cause all remaining subaction commit processing to be done, as described below.

6.6 Topaction Commit Process

The topaction commit process is almost identical to the topaction commit process in the fixed action model. A global ordering for topation validation is still required, and we use the same approach of having the coordinator choose a global queue entry time for its topaction. (Other alternatives were discussed in the Section 5.7.) The main difference here is that the shadow versions of the topaction are already located at the OM's where they will be installed as base versions. In the fixed action model, the shadow versions of the topaction were located at the topaction's AM, which sent these versions to the OM's in update sets as part of the commit process. The other major difference is that the coordinator includes the topaction's aborted set in `validate-request` messages.

6.6.1 Validation at a Coordinator

As in the fixed action model, the entity running the topaction (in this case an OM) acts as coordinator for the two-phase commit. The coordinator's role is shown in Figure 6.11. (This figure does not include some of the details of the coordinator's role in a two-phase commit,

```
Topaction T Attempts to Commit:
    global-qet(T) := a proposed queue entry time for T
    for each OM-P in participants(T) do
        send the message validate-request(id(T), rs(T), ws(T), aborted(T), global-qet(T))
                to OM-P
        end
    gather responses...
    if any response is FAILED then
            for each id(OM-P) in participants(T) do
                send the message abort(id(T)) to OM-P
                end
        else if any response is BAD-GLOBAL-TIME then
            global-qet(T) := a new proposed queue entry time for T
            for each id(OM-P) in participants(T) do
                send the message revalidate-request(id(T), global-qet(T)) to OM-P
                end
            go back to gather responses... above
        else if all responses are OK then
            for each OM-P in participants(T) do
                send the message commit(id(T)) to OM-P
                end
        end
```

Figure 6.11: Coordinator, Topaction Validation — Fixed Object Model

such as dealing with lost messages or recording the state of the commit in stable storage.) During the first phase, the coordinator sends `validate-request` messages to participating OM's; these messages include the topaction's read set, write set, and aborted set, and a global queue entry time. In the fixed action model, we computed the set of participants from the topaction's read set; for this model, we maintain an explicit participants set, so we do not need to do this computation. (The participants set is used for both subaction and topaction commit processing; it makes sense to keep an explicit set, rather than computing it each time from the read set.) Note that the topaction's OM also acts as a participant in the commit; of course, it does not send messages to itself, but otherwise it plays the same role as any other participant.

6.6.2 Validation at a Participant

A `subaction-commit` message can be lost, and a `validate-request` message for a topaction might arrive before the `subaction-commit` messages for some of its descendants arrive; thus, a participant may not have completed some subaction commit processing when a topaction's

```
Move Shadow Versions up to Topaction T:
    for each id(A) in aborted(T) do
        for each D in descendants(A) do
            abort D if it is still running
            status(D) := aborted
            end
        end
    for each A in proper-descendants(T) do
        if status(A) = active then
            for each existing shadow(A, id(O)) do
                shadow(T, id(O)) := shadow(A, id(O))
                end
            status(A) := committed
            end
        end
```

Figure 6.12: Moving Shadow Versions at Topaction Commit — Fixed Object Model

`validate-request` message arrives. Thus, there may be some shadow versions located at descendants of a committing topaction T that should have been moved up to T.

Before validating a topaction, we perform this shadow version movement; this is shown in Figure 6.12. First, the status of all aborted descendants of T is changed to ABORTED; some of these may be orphans that are still running, in which case they are aborted. The shadow versions of any active proper descendant of the topaction are then moved up to the topaction. This processing is essentially the same as that for a subaction commit. For the same reasons outlined in Section 6.5.3, it is safe to do this processing even if some `subaction-commit` messages have not been processed.

The processing of a `validate-request` message for topaction T is shown in Figure 6.13. This figure does not include some of the details of the participant's role in the two-phase commit process, such as acknowledging commit and abort messages or recording the state of the commit process in stable storage. As shown, T is validated after moving any shadow versions up to T. In fact, these two processes could occur concurrently.

The validation algorithm is similar to the topaction validation algorithm in the fixed action model (Figure 5.8). The participant returns an OK or FAILED response. If it responds with OK, and the coordinator sends a `commit` message, then the topaction's shadow versions are installed as base versions. Note that the topaction's shadow versions are already located at the participant; the `validate-request` message does not include an update set, as it did in the fixed action model.

6.7 Optimizations

This section outlines several optimizations for the fixed object model. Unlike the fixed action model, there is no optimization, such as caching, that radically changes the design of the model.

6.7.1 Discarding Information

Action information can be garbage collected. When action C at *OM-C* attempts to commit to parent P at *OM-P* by sending a `commit-request`, all information at *OM-C* about C except its shadow versions can be discarded after *OM-P* acknowledges the `commit-request`. We assume here that OM's send an acknowledgement in response to `commit-request` messages. After C commits or aborts at *OM-P*, all information about C can be discarded, since it is transferred to action P. When a `subaction-commit` message is processed, information about each action that is marked as committed or aborted can be discarded; *i.e.*, shadow versions are either be moved or discarded. When a topaction commits or aborts, all information about the topaction can be discarded after it has either installed its shadow versions as base versions or discarded its shadow versions. At each OM, the set *rw-top-committed* can be garbage collected as was described in Section 3.7.

6.7.2 Additional Message Ordering

The use of the *latest* data structure depends on message ordering. At *OM-X*, all `subaction-commit` messages sent from *OM-Y* with end times less than or equal to *latest(OM-Y)* have been processed. For this to hold, it must be true that `subaction-commit` messages are processed in the order of their end times; *i.e.*, in the order that they are sent from *OM-Y* to *OM-X*. A special ordered-message protocol (such as using sequence numbers) must be used to guarantee this.

There is an additional message-ordering constraint that OM's should guarantee to improve the chances that a subaction will validate successfully. If *OM-Y* sends a `start-subaction` message to *OM-X* *after* it sends a `subaction-commit` message to *OM-X*, then the subaction should be started after the `subaction-commit` is processed. Suppose a parent action starts a child action *C1* at *OM-C*, waits until it commits, and then starts another child *C2*, also at *OM-C*. Note that the `start-subaction` message for *C2* is sent

```
OM-X receives message:
      validate-request(id(T), rs(T), ws(T), aborted(T), global-qet(T))
   begin
   Move Shadow Versions up to Topaction T
      begin critical section for topactions
      if global-qet(T) $\leq$ qet then return BAD-GLOBAL-TIME end
      qet := global-qet(T)
      VQcopy := copy(VQ-TOP) ; rw-top-copy := copy(rw-top-committed)
      allocate entry E for action T; E.type := VALIDATING; add E to VQ-TOP
      end critical section for topactions
   st(T) := the earliest read time in rs(T)
   for each [ws(RW), et(RW)] pair in rw-top-copy with st(T) < et(RW) do
      for each [id(O), readtime-top] pair in rs(T) do
         if O is managed by this OM and
            id(O) $\in$ ws(RW) and readtime-top < et(RW) then exit FAIL end
         end
      end
   for each read-write action RW in VQcopy do
      for each [id(O), readtime-top] pair in rs(T) do
         if id(O) $\in$ ws(RW) then exit FAIL end
         end
      end
   return value OK
   wait for the message commit(id(T)) or abort(id(T))
   if the message is abort then exit FAIL end
   the message is commit...
      begin critical section for topactions
      E.type := READY
      while head(VQ-TOP).type = READY do
         let H be the action at the head of VQ-TOP
         remove H's entry from VQ-TOP
         if H is read-write then
            tnc-top := tnc-top + 1; et(H) := tnc-top
            OM-ws := OM-copy(ws(H), id(OM-X)) (copies only OM-X's objects)
            add the pair [OM-ws, et(H)] to rw-top-committed
            for each id(O) in OM-ws do
               base-version(id(O)) := shadow(H, id(O))
               end
            end
         (H is now a committed action.)
         end
      end critical section for topactions
   end
      except when FAIL:
               begin critical section for topactions
               delete E from VQ-TOP
               end critical section for topactions
            return value FAILED
            end
```

Figure 6.13: Participant, Topaction Validation — Fixed Object Model

to *OM-C* *after* the `subaction-commit` message for *C1*. If the `subaction-commit` message is not processed at *OM-C* before subaction *C2* starts, then *C2* will not see the changes made by *C1*, even though the parent action explicitly waited until *C1* committed before it started *C2*.

6.7.3 Message Buffering

It is possible to buffer `subaction-commit` messages. So far, we have considered all messages to follow the semantics of remote procedure call; when a message is invoked, we have assumed that it is sent immediately. While messages such as `start-subaction` messages or the messages sent during two-phase commit should be sent immediately, it is not necessary to send `subaction-commit` messages immediately. Instead, an OM can group together a number of `subaction-commit` messages destined for the same OM and use a single message for the set. In addition, an OM can "piggy-back" `subaction-commit` messages on messages that must be sent immediately. We can think of each OM as having a buffer for each other OM that it sends messages to. When it sends a `subaction-commit` message, it simply adds the message to the appropriate buffer; the buffer keeps message properly ordered. When it sends a message that must be sent immediately, it attaches the contents of the appropriate buffer to the message. As we described above, if the message being sent immediately is a `start-subaction` message, then any `subaction-commit` messages that get sent along with it should be processed before the subaction is started.

As usual, there is a tradeoff here. Message buffering improves communication performance, but it means that `subaction-commit` messages will arrive later than they would otherwise: some subactions will read older versions of objects, which can cause them to fail validation.

Chapter 7

Conclusion

In this chapter we summarize our work, compare our two distributed models, and outline a number of interesting areas for future research.

7.1 Summary

This thesis began by describing transaction systems and nested actions. Nested actions are a natural and desirable extension of top-level actions: they allow us to reason about concurrency within an action, and to divide an action into components that fail independently. We then discussed optimistic concurrency control, comparing it to pessimistic concurrency control, which is more widely used. We showed that, for systems with fixed action constraints and object caching, optimism should perform better than pessimism for accessing a cached object. Unfortunately, no current optimistic system supports nested actions, and the combination of optimism and subactions has not been studied in the literature. This thesis extends prior work in optimistic concurrency control to nested distributed actions. It begins by describing a model with nested actions in a single-site system, and then presents two distributed models with nested actions, the fixed action and fixed object models.

Our two models reflect the two different approaches that are taken when building transaction systems. In the fixed action model, an action brings copies of the objects that it accesses to the node where it is running; if it updates any of these copies, it sends the updates back as part of the topaction commit process. The most interesting research for this model was in the area of trying to reduce the number of object requests sent over the network. At this point we have only sketched out possible approaches: the most important approach will be to use object caching, while another important approach will be to use an object grouping mechanism to return more than one object copy in response to a single object request. To research the caching and object grouping strategies, it will be necessary to build a simulation.

In the fixed object model, all shadow versions of an object are kept at the same location as its base version. Most pessimistic systems have the same fixed object approach. The most interesting research for this model was in the area of trying to reduce the cost of subaction commit for non-local subactions. In general, the commit of an action requires an agreement protocol; *i.e.*, synchronization between two or more nodes in the system. For example, we use a two-phase commit protocol for topaction commit. For subactions, we would like to avoid the overhead associated with this kind of synchronization. This problem has been studied for pessimistic systems, and a viable solution has been found [Liskov *et al.* 1987a]. There would be a clear disadvantage to using optimism for fixed object systems if we did not have a method that avoids synchronization for subactions. Our approach, which uses asynchronous "courtesy" messages (messages that are not required to arrive), allows us to group together and "piggy-back" messages, significantly reducing network loading and the cost of subaction commit.

While our two models are not implementations, we do consider some efficiency issues and possible optimizations. From these discussions, it should be clear that either model could be used as the basis for a real system. In other words, including subactions in optimistic systems is not only desirable but feasible.

In systems following the fixed action model, optimistic concurrency control has an advantage over pessimistic concurrency control with respect to caching. In contrast, in the fixed object model (where there is no non-local object caching, since there are no non-local object copies), there are no clear reasons for choosing one concurrency control method over the other. In this sense, the fixed action model is more interesting for optimistic concurrency control. It is likely that future systems using optimism will be closer to the fixed action model than the fixed object model. However, the requirements of a given system will dictate which model is used.

7.2 Comparing the Distributed Models

In the fixed action model, a major issue is the cost of object movement. If objects are very large, or if actions frequently access a large number of objects, the cost of object movement will be high. Caching will help; in addition, one can use special methods to avoid the cost of transmitting large objects. For example, it is possible to transmit only relevant portions of large objects. Also, when sending commit information, we can send a list of changes to a large object instead of sending back the entire modified object.

One advantage of object movement is that it off-loads work from object managers. If objects are not moved, an object manager must perform all computation that is performed on its objects. In the fixed action model, the work is distributed to various action managers that can be located at different network nodes. In the fixed object model, if an OM is overloaded with work, we can distribute its set of objects over several OM's. However, this approach is not as flexible as the dynamic distribution of work in the fixed action model.

A potential problem in the fixed action model is security. Consider an employee database, where an employee is allowed to find out the average salary of the employees in the company, but not the salary of a specific employee. In the fixed object model, an object manager has complete control over all accesses to its own objects. Thus, it might respond to a request to compute the average, but not to a request for a particular salary. In the fixed action model, object state is passed out over the network to other locations. To compute the average, all employee objects would be passed out to the action manager doing the averaging — this may be unacceptable if the node running the averaging action is not considered secure.

This example raises another issue: locality. The fixed object model is designed to take advantage of *static locality*, while the fixed action model with caching is designed to take advantage of *dynamic locality*. Static locality is locality inherent in the placement of the objects in the system. We can place objects that are frequently accessed together at a single location. Dynamic locality is locality inherent in the computations that are performed by the system. An object that is accessed once tends to be accessed frequently by the same computation.

In the fixed object model, the location of objects is fixed. If objects can be statically arranged so that actions tend to use many objects at a single object manager, this will result in good performance. For example, if all employee objects are located at the same object manager, computing the average salary will have good performance. However, if the dynamic pattern of object accesses does not match the static placement of objects, performance will be significantly worse. In the fixed action model with caching, objects that are accessed frequently will be cached locally. If most employee objects are already cached, the computation of the employee average will have good performance. However, access to a large number of objects that are not accessed frequently (*e.g.*, computing the employee average when none of the employee objects is currently cached) will be expensive.

However, in addition to taking advantage of dynamic locality, the fixed action model can also attempt to take advantage of static locality. As was discussed in Section 5.7.4, an object manager can use grouping information to return a set of related objects in response to a single base-version request.

The relative performance of the two models depends heavily on many factors: the number, kind, and frequency of the actions in the system; object size; the number of objects accessed per action; object placement; caching strategy (for the fixed action model); and so on. For a given set of system requirements, one model may clearly be appropriate, but there will be cases when the choice is not clear — it may be necessary to run simulations to predict which model will be better.

There are a few distinctions between the models that do not deal directly with performance issues. For example, we did not mention the issue of code placement when describing the two models. If the objects in the system are typed objects with associated operations, it is necessary to have a local copy of the code for type T's operations to execute operations on some object O of type T. In the fixed object model, each object manager must have the code for its own objects' types. In the fixed action model, however, each action manager that accesses an object O must have the code for O's type. In the worst case, each AM will have to have the code for every type in the system.

One of the most interesting distinction between the two models is the cost of including subactions. First, imagine a fixed action model without subactions. We still have action managers and objects managers: AM's run topactions, while OM's do not change — they still respond to base-version requests and participate in the topaction commit process. The introduction of subactions merely entails local changes to the AM's, since subaction management, shadow versions management, and validation is all local. The messages sent in the system also do not change.

In contrast, imagine a fixed object model without subactions. Here, it is not necessary for an OM to keep transaction counters for other OM's, to send its own transaction counter in commit messages, or to record multistamps. Courtesy commit messages are not used — only topaction commits occur, so only full two-phase commit is used. To add subactions, every OM in the system must be modified: each OM must keep track of more information, send more information, and be able to process an additional message type (the courtesy commit). In the fixed action model it would be possible to add subactions to only some of the AM's; in the fixed object model, all OM's must add subactions at the same time.

7.3 Future Work

There is still much work to be done in the area of optimistic concurrency control. This work will be able to build upon what we have already studied. In particular, future work should explicitly state whether the results apply to fixed action or fixed object systems. In addition, it should not ignore subactions: something feasible (or infeasible) for topactions may prove to be infeasible (or feasible) for subactions, and in any case subaction processing will usually be different from the topaction processing. In this section, we outline some areas for future work.

7.3.1 Combining the Fixed Action and Fixed Object Models

A hybrid model that distributes both actions and shadow versions is probably possible. However, it would not be a trivial combination of the two models. For example, reading the nearest version of an object is non-trivial, since the nearest version might be at *any* ancestor's object manager.

7.3.2 Caching and Object Grouping Strategies

As we mentioned above, it will be necessary to build a simulation of the fixed action model to study various object caching and object grouping strategies. To drive such a simulator, traces from real systems will have to be obtained. Some objects ("hot spot" objects that are frequently modified) should *not* be cached — it should be possible for the system to dynamically detect such objects and mark them as "hot", as well as detect when they are no longer hot. Dynamic hot-spot detection is done at the page level in the Statice system [Gerson 1989]. Similarly, dynamic object grouping may be possible: the system can observe object access patterns to learn which objects should be grouped together.

For systems with integrated programming language support, the programmer may be able to help with caching (for example, by suggesting which objects will be hot spots), and will definitely be able to help with object grouping (by explicitly forming object groups). A number of groups have including object grouping mechanisms in their object-repository projects, including Observer [Hornick & Zdonik 1987] and Mneme [Moss & Sinofsky 1988].

7.3.3 Combining Optimism and Pessimism

In general, we would like to have a system that applies optimism and pessimism where they are most useful: optimism where synchronization is unacceptable, and pessimism where conflicts are likely.

A hybrid optimistic-pessimistic system would be interesting for both the fixed action and fixed object models. Imagine a system with two kinds of objects, optimistic and pessimistic ones, where actions acquire locks for pessimistic objects and record read set and write entries for optimistic objects. If an optimistic object is found to be a hot-spot, it can be changed to a pessimistic object.

In an optimistic-pessimistic fixed action model, when a base-version-request is processed, if the object requested was pessimistic it would be locked at its OM; if other base versions for pessimistic objects were also returned (because of an object grouping mechanism) we would not want to lock these objects. Thus, we can have pessimistic objects that are cached but are not locked. When accessing a cached pessimistic object that is not locked, an action can send the lock request before accessing the object; alternatively, it can concurrently access the object and send the lock request, as is done in Statice [Gerson 1989].

Thus, when caching is taken into account, there can be several categories of pessimistic objects. Once again, the system can attempt to determine dynamically what the appropriate category for an object is, or the programmer can specify the category if the system is integrated with a programming language.

7.3.4 Object Movement in Pessimistic Systems

Object movement and caching are interesting for optimistic systems; are they also interesting for pessimistic systems? It seems at first glance that a pessimistic version of the fixed action model with base version caching probably does not make sense because of strict cache coherency requirements. However, as the Statice system shows, it is possible to have a mostly pessimistic system with a hybrid approach to accessing cached objects.

For pessimistic systems similar to the fixed object model, a limited form of object movement may be interesting. Consider a system such as Argus [Liskov 1984] that executes remote procedure calls as subactions, where some or all of the parameters are passed by reference. To improve locality during the execution of the subaction, it may make sense to pass copies of some of the objects that are passed by reference. Moving objects to improve locality has been studied in Emerald [Hutchinson 1987] and REV [Stamos 1986]. Emerald has no transaction mechanism, and REV has only topactions with pessimistic concurrency control; it would be interesting to re-examine this work, considering pessimistic systems with nested actions. An important part of this work would be to determine when to move objects; again, the system may be able to learn this by monitoring its own performance, and the programmer may be able to help if appropriate linguistic constructs are provided.

7.3.5 Type-Specific Conflict Detection

Read-write conflict detection can lead to unnecessary action conflicts. If the objects are of abstract type, a type-specific conflict detection scheme can be used [Herlihy 1986]. We did not consider such a scheme, as it would have unnecessarily complicated our models. In addition, because of its extra overhead, it is not clear that switching to a type-specific scheme will result in improved performance. We believe that our models can be adapted to use type-specific methods, but we have not examined the issue in detail.

7.3.6 User-Defined Atomic Types

One way to provide type-specific conflict detection is to allow programmers to define their own atomic data types; this has been studied for Argus [Weihl 1984] and for Avalon, an extension to C++ [Herlihy & Wing 1987]. Neither study has considered support for optimistic objects. In Avalon, one develops user-defined (but system-invoked) commit and abort operations; to develop optimistic objects, it would also be necessary to have a validate operation.

7.3.7 Linguistic Issues

In addition to some of the linguistic issues already mentioned, there is the more basic issue of designing optimistic action constructs for a programming language. The usual *begin action, end action, commit,* and *abort* used for pessimistic actions may not be sufficient for optimistic actions. For example, what is the correct action to take if validation fails: do we want the system to restart the action automatically, or should the programmer specify what to do?

In a pessimistic system, the same kinds of issues arise. For example, an action can be aborted to resolve a deadlock, in which case it would normally be restarted automatically. Again, automatic restart is not appropriate in all cases.

A related issue is that of providing constructs to help programmers write actions that include user-visible input or output. It is often not possible to simply restart an action that involved user-visible operations. In an optimistic system, if an action that has user-visible operations conflicts with an action that does not, we would prefer that the system aborted the latter action. Similarly, in a deadlock situation, we want a pessimistic system try to avoid aborting an action with user-visible operations.

7.3.8 Orphans

An *orphan* is a computation that continues to run even though its results are no longer needed. In an optimistic concurrency control system, orphans arise in three ways: from aborts, commits, and crashes. Abort orphans and crash orphans occur in both optimistic and pessimistic systems, and have been studied previously [Liskov *et al.* 1987b]. Commit orphans are unique to optimistic systems.

If a parent action aborts (or its node crashes) while its child is still executing at another entity, the child action is an abort orphan: it continues to execute, but its results are not needed, and will be ignored if they are returned to the entity of the parent. When an action commits, if a concurrent sibling has read a set of objects that overlaps with the write set of the committed action, then the sibling is a commit orphan: it will fail validation, and its results will be thrown away. If an object manager that is managing one or more objects that have been modified by an action crashes while the action is still active, then the action is a crash orphan: it will not be able to commit through the top, since it will not be able to install its changes at the crashed object manager.

Orphans fail when they try to commit; however, it is desirable to abort orphans as soon as possible, for two reasons: they waste resources such as processor time, and, if they perform user-visible operations, they can present inconsistent information to the user, since they are not serialized with respect to committed actions. The process of detecting and destroying orphans is commonly referred to as *orphan detection.* If orphans are always detected before they can see an inconsistent state, this solves one of the major drawbacks of optimistic systems: namely, the fact there are additional orphans that can see inconsistent state. (See Section 7.3.9 for another approach to this inconsistent state problem.) Unfortunately, orphan detection itself has overhead, and tends to slow down the processing of all actions (both orphans and non-orphans) throughout the system. It is an open question whether orphan detection actually improves performance.

The models in this thesis use backward validation, where actions validate against already-committed actions. Another possibility is to use forward validation, where actions validate against *active* actions. A major advantage of forward validation is that it does not create commit orphans, since an action is only allowed to commit if it does not invalidate active actions. However, forward validation does not make sense for topactions: it would be necessary for an action's siblings to block while it validates and commits, and the two-phase validation and commit process is too lengthy to consider blocking active topactions during topaction validation. Forward validation for subactions might be useful, however. It would be especially easy to use forward validation for subactions in the fixed action model, since this validation is local to a single action manager.

7.3.9 Multiversion Schemes

Some systems have explored the possibility of keeping a timestamped sequence of base versions for each object, where the timestamps indicate the time that the versions were committed, and the version with the most recent timestamp is the current value of the object. In these systems, it is possible to separate the commit processing of read-only actions from the commit processing of read-write actions. When a read-only action reads an object O, it reads the version of O with the largest timestamp that is less than or equal to its start time. We can picture the read-only action as taking a "snapshot" of all of the objects in the system at the time it begins, and then reading objects from that snapshot. Such an action can always be serialized with respect to all other actions in the system. Thus, in an optimistic system with multiple versions, it is not necessary to validate read-only actions; read-only actions always commit, and read-write actions only need to validate against other read-write actions. An additional advantage of this approach is that actions always see a consistent state, since they are reading from a consistent "snapshot" of the objects in the system

An optimistic multiversion scheme has already been proposed [Agrawal *et al.* 1987]. However, this work only considers a fixed object approach without subactions. In addition, some aspects of the approach need to explored in more depth. For example, if a read-write action uses its start time to select object versions, this ensures that the action will see a consistent state, but makes it more likely that it will read older versions of objects and end up failing validation. For actions that perform user-visible operations, viewing a consistent state is essential; perhaps two kinds of actions should be provided, one that ensures a consistent view and one that does not. Actions that do not ensure a consistent view would always read the most recent versions of objects.

7.3.10 Concurrency Control in Shared-Memory Multiprocessors

Small-scale shared-memory multiprocessors can be considered physical instances of the fixed action model: the distributed memories correspond to object managers, the processors correspond to action managers, and the caches at the processors correspond to base version caches at action managers. Currently, shared-memory multiprocessors maintain strict cache coherence, using a "snoopy" cache-coherency scheme such as the Dragon scheme [McCreight 1984] that maintains consistency across each read and write that occurs at any processor. However, it can be argued that concurrent programs for these machines will be written using some sort of concurrency control mechanism, such as critical sections; therefore, coherency only needs to be maintained at certain points in time, such as at the end of a critical section, before the section's semaphore is released.

Rather than using critical sections, one can imagine using transactions. Can traditional optimistic or pessimistic techniques normally used for distributed systems be applied to this domain?

References

[Agrawal 1983] R. Agrawal. *Concurrency Control and Recovery in Multiprocessor Database Machines: Design and Performance Evaluation.* Ph.D. thesis, University of Wisconsin, 1983.

[Agrawal *et al.* 1987] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed optimistic concurrency control with reduced rollback. *Journal of Distributed Computing, Springer-Verlag*, 2(1):45–59, 1987.

[Badal 1981] D. Z. Badal. Concurrency control overhead or a closer look at blocking vs. non-blocking concurrency control mechanisms. In *Proceedings of the 5th Berkeley Workshop*, pages 55–103, 1981.

[Carey 1983] M. Carey. *Modeling and Evaluation of Database Concurency Control Algorithms.* Ph.D. thesis, University of California, Berkeley, September 1983.

[Ceri & Owicki 1982] S. Ceri and S. Owicki. On the use of optimistic methods for concurrency control in distributed databases. In *Proceedings of the 6th Berkeley Workshop*, pages 117–130, 1982.

[Cohen 1989] Jeffrey I. Cohen. *Atomic Stable Storage Across a Network.* Master's thesis, Massachusetts Institute of Technology, May 1989.

[Daniels *et al.* 1987] Dean S. Daniels, Alfred Z. Spector, and Dean S. Thompson. Distributed logging for transaction processing. In *ACM Special Interest Group on Management of Data 1987 Annual Conference*, pages 82–96, ACM SIGMOD, May 1987.

[Davies 1978] C. T. Davies. Data processing spheres of control. *IBM Systems Jounal*, 17(2):179–198, 1978.

[Dwork & Skeen 1983] Cynthia Dwork and Dale Skeen. The inherent cost of nonblocking commitment. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 1–11, ACM, August 1983.

[Franaszek & Robinson 1985] P. Franaszek and J. T. Robinson. Limitations of concurrency in transaction processing. *ACM Transactions on Database Systems*, 10(1):1–28, March 1985.

[Gerson 1989] Dan Gerson. May 1989. Private communication.

[Gray 1979] J. N. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, chapter 3.F, pages 394–481, Springer-Verlag, 1979.

[Harder 1984] T. Harder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, June 1984.

[Herlihy & Wing 1987] Maurice P. Herlihy and Jeannette M. Wing. Avalon: language support for reliable distributed systems. In *Proceedings of Seventeenth International Symposium on Fault-Tolerant Computing*, pages 89–94, IEEE, Pittsburgh, March 1987.

[Herlihy 1986] Maurice Herlihy. Optimistic concurrency control for abstract data types. In *Proceedings of the Fifth Annual Symposium on Principles of Distributed Computing*, pages 206–217, ACM, August 1986.

[Hornick & Zdonik 1987] Mark F. Hornick and Stanley B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Information Systems*, 5(1):70–95, 1987.

[Hutchinson 1987] Norman. C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* Ph.D. thesis, University of Washington, January 1987. Also available as Department of Computer Science technical report 87-01-01.

[Kung & Robinson 1981] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[Lamport 1978] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lampson 1981] Butler Lampson. Atomic transactions. In B. W. Lampson, M. Paul, and H. J. Siegert, editors, *Distributed Systems — Architecture and Implementation*, chapter 11, pages 247–265, Springer-Verlag, 1981. Lecture Notes in Computer Science series, number 105.

[Lausen 1982] G. Lausen. Concurrency control in database systems: a step towards the integration of optimistic methods and locking. *Proceedings of the ACM*, 1982.

[Lausen 1983] G. Lausen. Formal aspects of optimistic concurrency control in a multiversion database system. *Information Systems*, 8(4):291–301, 1983.

[Lindsay *et al.* 1979] B. G. Lindsay, P. G. Sellinger, C. Galtieri, J. N. Gray, T. G. Price, F. Putzolu, I. L. Traiger, and B. W. Wade. *Notes on Distributed Databases.* Research Report RJ2571, IBM, San Jose, CA, July 1979.

[Lindsay *et al.* 1984] B. G. Lindsay, L. M. Haas, P. F. Wilms, and R. A. Yost. Computation and communication in r*: a distributed database manager. *ACM Transactions on Computer Systems*, 2(1), February 1984. Also in Proceedings of the 9th ACM Symposium on Operating Systems Principles, October 1983. Also available as IBM Research Report RJ3740, January 1983.

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

[Liskov *et al.* 1987a] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of argus. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, November 1987. Also available as Programming Methodology Group Memo 57, MIT Laboratory for Computer Science, Cambridge, MA, August 1987.

[Liskov *et al.* 1987b] Barbara Liskov, Robert Scheifler, Edward Walker, and William Weihl. *Orphan Detection.* Programming Methodology Group Memo 53, MIT Laboratory for Computer Science, Cambridge, MA, February 1987.

[Lynch *et al.* 1988] Nancy Lynch, Michael Merritt, Willaim Weihl, and Alan Fekete. *The Theory of Atomic Transactions.* Technical Memo TM-362, MIT Laboratory for Computer Science, Cambridge, MA, June 1988.

[Maier *et al.* 1986] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an object-oriented dbms. *Sigplan Notices: OOPSLA-86 Conference Proceedings*, 21(11):472–482, November 1986.

[McCreight 1984] E. McCreight. *The Dragon Computer System: An Early Overview.* Technical Report, Xerox Corporation, September 1984.

[Menasce & Nakanishi 1982] D. A. Menasce and N. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1):13–27, 1982.

[Mills 1988] D. L. Mills. *Network Time Protocol (version1) Specification and Implementation.* DARPA-Internet Report RFC-1059, DARPA, July 1988.

[Mohan & Lindsay 1983] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 76–88, ACM, August 1983.

[Moss & Sinofsky 1988] J. Elliot B. Moss and Steven Sinofsky. *Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface.* COINS Technical Report 88-67, Object Oriented Systems Laboratory, Department of Computer and Information Science, University of Massachusetts, Amherst, Amherst, MA, July 1988.

[Moss 1981] J. Elliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* Technical Report 260, MIT Laboratory for Computer Science, Cambridge, MA, April 1981.

[Perl 1988] Sharon E. Perl. *Distributed Commit Protocols for Nested Atomic Actions.* Master's thesis, Massachusetts Institute of Technology, February 1988.

[Reed 1978] D. P. Reed. *Naming and Synchronization in a Decentralized Computer System.* Technical Report 205, MIT Laboratory for Computer Science, Cambridge, MA, 1978.

[Reed 1983] David Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[Skeen 1981] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the International Conference on Management of Data*, pages 133–142, ACM/SIGMOD, 1981.

[Spector & Swedlow 1987] Alfred Z. Spector and Kathryn R. Swedlow. *Guide to the Camelot Distributed Transaction Facility: Release 1.* Technical Report, Computer Science Department, Carnegie Mellon Univerity, Pittsburgh, PA, September 1987. Draft: 0.7(31)[aleph] edition.

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

[Stamos 1986] James W. Stamos. *Remote Evaluation.* Ph.D. thesis, Massachusetts Institute of Technology, January 1986. Also available as Laboratory for Computer Science Technical Report MIT/LCS/TR-354, Cambridge, MA, January 1986.

[Tay *et al.* 1984] Y. C. Tay, N. Goodman, and R. Suri. *Performance Evaluation of Locking in Databases: a Survey.* Technical Report RT-17-84, Harvard Aiken Laboratory, 1984.

[Thomas 1978] R. H. Thomas. A solution to the concurrency control problem for multiple copy databases. In *Proceedings of the 16th IEEE Computer Society Internation Conference (COMPCON)*, Spring 1978.

[Weihl 1984] William E. Weihl. *Specification and Implementation of Atomic Data Types.* Technical Report 314, MIT Laboratory for Computer Science, 1984.

[Weihl 1986] William Weihl. *Distributed Version Management for Read-Only Actions.* Programming Methodology Group Memo 47, MIT Laboratory for Computer Science, Cambridge, MA, May 1986.

[Weinreb *et al.* 1988] Daniel Weinreb, Neal Feinberg, Dan Gerson, and Charles Lamb. An object oriented database system to support an integrated programming environment. *IEEE Data Engineering*, 11(2):33–43, 1988.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-453	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science	6b. OFFICE SYMBOL *(If applicable)*
7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy	
6c. ADDRESS *(City, State, and ZIP Code)* 545 Technology Square Cambridge, MA 02139	7b. ADDRESS *(City, State, and ZIP Code)* Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD	8b. OFFICE SYMBOL *(If applicable)*
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS *(City, State, and ZIP Code)* 1400 Wilson Boulevard Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Optimistic Concurrency Control for Nested Distributed Transactions

12. PERSONAL AUTHOR(S)
Gruber, Robert Edward

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical	FROM ______ TO ______	1989 June	106

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Concurrency control, Optimistic concurrency control, Transactions, Nested transactions, Atomicity, Commit protocols, Distributed computer systems

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

Optimistic concurrency control techniques allow atomic transactions (or actions for short) to execute without synchronization, relying on commit-time validation to ensure serializability. Previous work in this area has focussed on single-level actions. This thesis extends prevous work on optimistic concurrency control to distributed systems with nested actions.

The thesis presents two contrasting models for managing nested actions, which we call the fixed actions model and the fixed object model. In the fixed action model, an action executes at only a single network node; if an action accesses an object whose storage is provided by another node, it brings a copy of the object to its node. In this model, caching copies of non-local objects can be used to improve performance by reducing the number of non-local object requests. We show that optimistic concurrency control has an advantage over pessimistic concurrency control with respect to this object caching. In the fixed object model, actions can span network nodes: an action at one node can start a nested

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE *(Include Area Code)* (617) 253-5894
22c. OFFICE SYMBOL	

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted.
All other editions are obsolete.   SECURITY CLASSIFICATION OF THIS PAGE

☆U.S. Government Printing Office: 1985—507-047

Unclassified

19. action at another node. However, an object's state is never moved from the node providing its storage. While there is a clear reason for using optimistic concurrency control for systems following the fixed action model, there is no clear reason for choosing either optimism or pessimism for systems following the fixed object model, however, the development of this model is an important step in the study of hybrid models.