MIT/LCS/TR-423

VIEWSTAMPED REPLICATION FOR HIGHLY AVAILABLE DISTRIBUTED SYSTEMS

Brian Masao Oki

August 1988

Viewstamped Replication for Highly Available Distributed Systems

by

Brian Masao Oki

August 1988

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 grants DCR-8503662 and DCR-8510014.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts

Viewstamped Replication for Highly Available Distributed Systems

by

Brian Masao Oki

Abstract

This dissertation presents *viewstamped replication*, a new algorithm for the implementation of highly available computer services that continue to be usable in spite of node crashes and network partitions. Our goal is to design an efficient mechanism that makes it easy for programmers to implement these services without complicating the programming model. Our replication method is based on a primary copy technique, where one replica is the primary and others are backups, and is integrated into the fabric of an atomic transaction mechanism. Transactions are run only at the primary and need not involve the backups; the primary propagates the effects of transaction processing to the backups in the background. The method exhibits low delay during normal operation, has low overhead, and increases the likelihood that transactions will commit in spite of failures.

When failures occur, replicas are reorganized automatically and a new primary is selected if the old one becomes inaccessible. This reorganization is called a view change and is accomplished by a *view management algorithm*. Since the primary only communicates with the backups in background mode, the effects of some processing may be lost after a view change; the affected transactions must abort. If the effects are known at the new primary, then no information is lost and the transaction can commit. Furthermore, if transactions commit, we guarantee that their effects are not lost. A special kind of timestamp, called a *viewstamp*, allows the algorithm to distinguish these cases inexpensively.

Keywords: Primary copy, Replication, Viewstamp, View management, High availability, Fault-tolerance, Transactions, Nested transactions, Atomicity, Distributed computer systems

This report is a minor revision of a dissertation of the same title submitted to the Department of Electrical Engineering and Computer Science on May 20, 1988 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science. The dissertation was supervised by Professor Barbara H. Liskov.

Contents

Figures

Acknowledgments

My years at MIT have been a revealing voyage of self-discovery through the hardships and triumphs of graduate school.

I thank my thesis supervisor, Barbara Liskov, for reading countless drafts of this dissertation and for signficantly improving its presentation. I am grateful to her for teaching me how to do research and how to write. I thank my thesis readers, Dave Gifford and Bill Weihl, for reading quickly the draft I submitted for my defense and providing me with invaluable comments. I am grateful to Dave for his wise counsel, constant support, and encouragement. He listened patiently when I needed to talk. Bill deserves many thanks for innumerable kindnesses, both while we were fellow graduate students and later as a thesis reader. Shrewd and wise John Guttag provided much sound advice about job interviewing.

I thank Debbie Hwang and Sanjay Ghemawat, my officemates, for goodnaturedly tolerating my grumpy moods and occasional sour demeanor, especially during the final few weeks as I wrote and revised furiously in order to finish. I thank Debbie for many enlightening conversations, technical and otherwise. My thanks go to her for helping me to work out a way of repairing my view management algorithm, which did the wrong thing in the presence of any sequence of crashes and partitions. I thank group members, past and present—Boaz Ben-Zvi, Mark Day, Ken Goldman, Elliot Kolodner, Rivka Ladin, Gary Leavens, Eliot Moss, Sharon Perl, Liuba Shrira, Ed Walker, and Andrew Xu—for their friendship and for serving as sounding boards for my ideas. Special thanks to Gary Leavens for his support during trying times. To those who read my thesis, in whole and in part, I thank you for your comments and suggestions: Alan Fekete, Sanjay, Debbie, Elliot, Gary, Liuba, and Andrew.

I thank Maurice Herlihy for his practical suggestions regarding the kinds of thesis topics that one should do and that one should avoid. Craig Schaffert advised me to abandon my original Ph.D. thesis topic with cogent, compelling, and utterly sound arguments. I'm glad I listened to him. I thank Sheng-Yang Chiu, ex-officemate, for his friendship and moral support. Brian Coan was responsible for my taking up bicycling and joining the 5th floor bikers on various trips and going to Talbot House. Jennifer Welch, Kathy Yelick, and Jeannette Wing understood me and listened when I needed to talk, and I am grateful to them. I thank my friend, Janice Chung, not only for understanding me, but also for putting up with my horrible leads when we both learned to dance Swing four years ago.

Gracious, supportive, and always encouraging, Jim Gray of Tandem told me over lunch about his years in graduate school at UC Berkeley and his ideas on how one ought to go about doing a doctoral thesis. I am grateful to him for listening to my ideas and asking hard questions. I thank Pat Helland of Tandem for taking the time to answer my questions about Tandem's commit protocol a few days before I submitted the dissertation; I hope I got it right. From UC Irvine's Tim Standish and George Lueker, I learned a lot about data structures and the analysis of algorithms, and had a fleeting glimpse of the world of research.

To the Ashdown gang, for several years of camaraderie and conversation: Sam Cooper, Ziyad Durón, Eugene Gath, Glen Kissel, Monty McGovern, Mark Smith, and Robin Vaughan. I thank Nancy Lee for her friendship and for the many hours of delightful companionship and conversation we've had. She has tried to teach me to relax and to enjoy life more, and helped make my last semester at MIT the happiest ever. I thank my friends—Bob Gounley, Rob Granville, Jim Restivo, and Su-Ming Wu—whose moral support and friendship over the years made my stay at MIT more enjoyable.

Ballroom dancing has been a welcome and fun diversion for me over the last five years from the labors of graduate school. My thanks go to Ron Gursky, Dan Radler, and Suzanne Hamby, for teaching me how to dance the international style Latin-American and Modern ballroom dances.

Finally, I thank Mom and Dad and my sister, Lisa, for always believing in me, inspiring me to do the best I was capable of doing, and never doubting that I would finish.

1 Introduction

High availability is essential to many computer-based services. For example, imagine trying to withdraw money from your savings account at your local savings and loan bank. As you present your passbook to the teller, he tells you, red-faced with embarrassment, that he cannot process your request because the system is down and will be for some time. I actually found myself in the teller's position some years ago, apologizing profusely to customers. The entire customer database resided on a single mainframe computer located somewhere in Beverly Hills, California. All branches in Los Angeles County were connected directly to this computer, and its problems affected everybody. In a fit of rage, some customers later closed out their accounts and took their money elsewhere, much to the bank's chagrin.

Consider another example, TWA's airline reservation system [Gifford 84]. Reservation agents in Boston, Chicago, and Los Angeles might be booking flights on behalf of customers and accessing the monolithic reservation system from different parts of the country. If the reservation system resided on a single computer, a failure of that computer could immobolize the entire airline because the airline is critically dependent on the system. An agent is Chicago, for example, would be unable to book flights because the system is unavailable. This is a bad situation for the airline to be in because it will definitely lose revenue and possibly passenger goodwill.

The key to achieving high availability is replication. Historically, the idea was to replicate each hardware component to such an extent that the likelihood of all replicas failing became vanishingly small [von Neumann 56]. Likewise, we can replicate important information at several computers. Flight information, for example, is more likely to be usable by a reservation agent if there are several copies, because the failure of a single computer leaves other copies available. We say that a service is *highly available* if it continues to be usable with high probability in spite of failures.

This dissertation addresses the problem of constructing highly available computer-based services that automatically tolerate computer crashes and communication failures. Ideally, programmers would write programs to implement these services in some distributed programming language that supports our model of computation, without worrying about availability. Our goal is to design a mechanism that can be used by a language implementation to take care of the details of implementing availability automatically.

This dissertation presents a new replication method. Our method operates in a distributed computer system that consists of many nodes connected by a communication network. Distinct nodes communicate with each other only by sending messages over the network. Nodes are individual computers, and may be uniprocessors, multiprocessors, timesharing machines, or single-user workstations.

Both the nodes and the network may fail; we assume these failures are not Byzantine [Lamport 82]. Nodes may crash, but we assume they are failstop [Schneider 83], that is, they fail by halting. Each node has volatile storage that is lost in a crash. The network may lose, duplicate, or delay messages, or deliver them out of order. Link failures may cause the network to partition into isolated subnetworks that cannot communicate with one another. We assume that failures are eventually repaired: nodes eventually recover from crashes and partitions are eventually reconnected.

We have integrated our replication method into a system that supports transactions. We assume that programmers write distributed programs consisting of *modules*, each of which resides at a single node in the network. *Atomic transactions* [Eswaran 76] perform computations involving these modules. Each module contains within it both data objects and code that manipulates the objects; modules can recover from crashes with some of their state intact. No other module can access the data objects of another module directly. Instead, each module provides procedures that can be used to access its objects;

modules communicate by means of remote procedure calls (RPCs) [Nelson 81]. Atomic transactions guarantee serializable and recoverable execution, preserving consistency of the data in the presence of failures and concurrent activities. Our scheme also works with nested transctions [Moss 81, Davies 78]. An action that is nested inside another is called a *subaction*.

Any replication method must solve two problems: maintaining the consistency of replicated data and synchronizing concurrent operations on the replicated data. Our method solves both problems and thus guarantees the *one-copy serializability* correctness criterion [Bernstein 83, Papadimitriou 79]. That is, the concurrent execution of transactions on replicated data is equivalent to a serial execution on non-replicated data.

1.1 Contributions

This dissertation makes several contributions.

The first contribution is *viewstamped replication*, a new primary copy replication algorithm, which was inspired by Alsberg and Day's primary copy technique [Alsberg 76]. The module is the unit of replication in a distributed program. Each replicated module consists of several instances, called *cohorts*, constituting a module group. One cohort is designated the primary; the others are backups. The primary is responsible for the module group's activity; it executes remote procedure calls and modifies its state. After a call executes, the primary propagates the effects of the call to the backups in background mode asynchronously. In this dissertation, we have worked out the details of a primary copy method, capitalizing on its advantages that other methods, such as weighted voting [Gifford 79] and quorum consensus [Herlihy 86], have not exploited. The advantages are as follows. First, the primary can be placed at an advantageous location, for example, at the main place of use or where there is a more powerful computer. Second, there are no synchronization problems because remote procedure calls are executed entirely at the primary and need not involve synchronizing with the backups. Third, the method exhibits low delay since users need only communicate with the primary copy.

The second contribution is a *view management algorithm* that limits the impact of node crashes and communication failures efficiently. Our algorithm is a simplification and modification of El Abbadi *et al.'s* virtual partitions protocol [El Abbadi 85]. When nodes or communication links fail and then recover, the cohorts are reorganized and a new primary is selected if the old one becomes inaccessible. Following El Abbadi and Toueg [El Abbadi 86], we refer to this reorganization as a *view change*. Once the view change is complete, the module group can continue to be used.

When a module group undergoes a view change, the effects of any remote procedure calls that ran on behalf of a transaction at the primary of the old view may or may not survive into the new view. We use a special kind of timestamp called a *viewstamp* to determine inexpensively whether needed information did survive. If the effects of calls survived, the transaction can commit; otherwise, it must abort.

The third contribution is integrating the replication method with transactions, rather than building it on top of a transaction mechanism, in contrast to most methods. This integration is beneficial in two ways. First, it is efficient because the replication method can take advantage of messages that must flow as part of transaction processing; messages can thus serve a dual purpose. Second, it allows us to fine-tune the system to varying levels of performance and availability. We can trade off computation at the primary in exchange for reducing the probability of aborting transactions. At one end of the continuum of possibilities, the primary might perform more computations, eagerly propagating updates to its backups when remote procedure calls complete. Although performance may suffer, we have reduced the probability that the transaction will abort; when the call returns, we can be certain its effects will not be lost in a view change. At the other end, the primary might do very little and let the updates propagate lazily after calls complete. If we are too lazy, however, transactions may abort more often.

The fourth contribution is that our method is an experiment in using the backups to record committed information instead of stable storage[1] as in a conventional transaction system. In such a conventional system the effects of committed transactions are made permanent by writing information at commit time to stable storage. In a replicated system, however, it seems redundant and expensive to record information both at the backups and on stable storage. In our method we assume that most of a cohort's state is volatile. Such an assumption means that a module group can potentially lose information about the effects of committed transactions if certain catastrophes befall the group. We

[1]Stable storage is an abstraction of a reliable memory device that does not lose information entrusted to it with high probability [Lampson 81].

analyze the effect of our method with respect to catastrophes.

1.2 Roadmap

We developed our replication method in the context of the Argus language and system [Liskov 88], which is described in Chapter 2. To make our discussion in later chapters more concrete we draw heavily on the Argus terminology. The Argus model is an instance of the general transaction model, however, and our method can be applied to any system with the same properties. Distributed programs written in Argus consist of guardians (corresponding to modules) and atomic actions (transactions).

In Chapter 3, we present an overview of our replication method and discuss how it interacts closely with transaction processing. In particular, we introduce the concepts of *view*, *viewstamp*, and *viewstamp history*. A view consists of a primary and its backups, which must constitute a majority of cohorts of a group; it represents the shared belief of cohorts in a group about who is accessible. The view management algorithm reorganizes the cohorts to form new views under certain conditions. Viewstamps represent how much a cohort "knows" about the effects of transactions that have run. The viewstamp history represents the sequence of view changes seen by a cohort; the cohort state reflects the effects of all events that happened in all views in the history. What a cohort *does* know and what it *should* know are used to determine whether transactions can commit or must abort. We state informally the conditions that transaction processing and our replication and view management algorithms must satisfy for correct operation of the system.

In Chapter 4, we describe in detail how to integrate our replication method into a transaction system. We use the Argus implementation as the basis for this work. In particular, we describe how our method works with both nested and flat transactions. We show that nested actions are useful to further reduce the probability of transaction abort by masking view changes.

In Chapter 5, we present our view management algorithm. In response to a failure, the algorithm reorganizes the cohorts, and if a majority agree to the change, it chooses a new primary and assembles a new view. The algorithm guarantees that the effects of all committed transactions survive successive view changes. If every view consists of at least a majority of cohorts of a group, then it must contain at least one cohort that knows about committed transactions in the previous view. What that cohort knows is used to bring all other cohorts in the new view up to date. The algorithm relies on properties of transaction commit to ensure that the effects of committed transactions are recorded at all the backups in the current view.

In Chapter 6, we discuss how our method compares with other replication techniques.

In Chapter 7, we conclude with a summary of what we have accomplished and a discussion of directions for future research.

2 Argus Programming Language and System

Argus is an integrated programming language and system [Liskov 88] that supports the implementation and execution of distributed programs. Distribution gives rise to some problems that either do not exist in a centralized system or exist in a less complex form. For example, a centralized system is either running or crashed, but a distributed system may be partly running and partly crashed. Argus provides mechanisms that help programmers cope with these problems.

Argus is intended to be used primarily for programs that maintain on-line data for long periods of time, such as file systems, mail systems, and banking systems. These programs require on-line information to remain consistent in spite of failures and also in spite of concurrent access. Programmers may need to control placement of information and processing at nodes to improve performance, since information is cheaper to access if it is nearby. Finally, programs may need to be reconfigured dynamically, for example, by adding and removing components, or by moving a component from one node to another.

In this chapter, we present the Argus model of computation. We focus on guardians and atomic actions.

2.1 Guardians

Argus modules are called *guardians*. An Argus guardian is a special kind of abstract data object that encapsulates and controls access to resources, such as databases or devices. It permits its resources to be accessed by means of special procedures, called *handlers*, that can be called from other guardians. For example, a guardian might encapsulate some or all of the accounts at a bank branch, and provide handlers to open and close accounts, and to withdraw and deposit money in accounts. As another example, a guardian might control a printing device, and provide a handler called *enq* to allow files to be enqueued for printing and a handler called *check_queue* to check the state of the queue.

Guardians are the logical nodes of the system. Each guardian resides at a single physical node of the network and may not span node boundaries. A node, however, may support several guardians.

A guardian contains within it data objects that store the state of its resource. These objects are not accessible outside the guardian; the only way they can be accessed or modified by another guardian is by calls of their guardian's handlers. Handler calls are *remote procedure calls* [Nelson 81]. The caller supplies the name of the called handler and some arguments. When the handler returns, the caller receives the results and can then continue processing. Arguments and results are passed by value. This rule ensures that a guardian retains control of its own objects and cannot be accessed directly by any other guardian. The Argus implementation takes care of all details of constructing and sending messages.

Inside a guardian are one or more processes. These processes can access all of the guardian's objects directly. Some processes carry out handler calls; whenever a handler call arrives at a guardian, a process is created to run the call. In addition, there may be *background* processes that carry out tasks independently of particular handler calls. For example, the *enq* handler of the printer guardian might merely record information about the request; a background process would carry out the actual printing.

A guardian is resilient to failures of its node. After a crash and recovery of its node, the guardian can recover with its code and objects intact. The objects have the values they possessed as of the last time they were written to stable storage. Stable storage preserves information written to it with very high probability [Lampson 81]. The objects in the guardian state are of two kinds: stable and volatile; only the stable objects survive crashes.

A crash destroys all volatile objects of a guardian and also all processes that were running at the time of the crash. After the crash, the Argus system restores the guardian's code and recovers the stable objects from stable storage. Then it creates a special recovery process, which runs code defined by the guardian to initialize the volatile objects. When this finishes, the guardian is ready to accept new handler calls and to run background processes. Since the volatile state does not survive crashes, it should be used only to record redundant information, such as an index into a database, or information that can be discarded in a crash, such as current printing information in the printer spooler. For example, in the printer guardian, information about queued requests would be stored in stable objects so that requests are not lost in a crash. Detailed information about the exact processing of the current request need not be stable, however, since the request can be redone after a crash.

A guardian can create other guardians dynamically, and the names of guardians and handlers can be sent as arguments of handler calls. A programmer can specify the node at which the new guardian is to reside; in this way individual guardians can be placed at the most advantageous locations. Handler calls are location-independent, so that one guardian can use another without knowing its location.

2.2 Atomic Actions

To solve the problems of concurrency and failure, computations in Argus run as *atomic transactions*, or *actions* for short. Atomic transactions have two properties. First, they are *serializable*, that is, the effect of running a group of actions is the same as if they were run sequentially in some order. Serializability permits concurrent execution, but ensures that concurrent actions cannot interfere with one another. Second, actions are *total*, that is, an action either completes entirely or it is guaranteed to have no visible effect. An action that completes is said to *commit*; otherwise, the action *aborts*. In addition, the effects of committed actions are *permanent*, that is, the effects persist and are guaranteed to survive failures.

Acquiring a read lock:

- All holders of write locks on X must be ancestors of S.

Acquiring a write lock:

- All holders of read and write locks on X must be ancestors of S.
- If this is the first time S has acquired a write lock on X, push a copy of the object on top of the version stack.

Commit:

- S's parent acquires S's lock on X.
- If S holds a write lock on X, then S's version (which is on the top of the version stack for X) becomes S's parent's version.

Abort:

- S's lock and version (if any) are discarded.

Figure 2.1: Rules for locking and version management for a subaction S, on object X.

are summarized in Figure 2.1. A subaction can acquire a read lock only if all holders of write locks are ancestors (that is, itself, its parent, its parent's parent, and so on); it can acquire a write lock only if all holders of read or write locks are ancestors, and in this case a new version is created for its use the first time it acquires a write lock. When a subaction aborts, its locks and versions are discarded and its parent action can continue from the state at which the subaction started. If a subaction commits, its locks and versions are inherited by its parent. If the parent aborts later, all modifications of the subaction will be undone. The rules make sense because Argus does not permit a parent to run concurrently with its children, neither does it permit any concurrency within an action except by creating subactions. For example, if a parent could run concurrently with a child, then the commit of the child could overwrite changes made by the parent since the child was created. The rules are implemented by a stack of versions, one version for each active action that is modifiying the object. When a subaction needs a new version, the version on top of the version stack is copied and the result pushed on the stack.

Every handler call runs as a subaction; this subaction runs on the caller's side and is called the *call action*. This extra action ensures that calls have a *zero or one* semantics:

If the call is successful and the called guardian returns a reply, it is guaranteed that the call happened exactly once. If it is not possible to complete the call, we abort the call action, thus guaranteeing that the call, effectively, did not happen at all. Running a call as a subaction ensures that calls have a clean semantics, which is a non-trivial and desirable property in a distributed system.

The processing of a handler call at the called guardian runs as a subaction of the call action; this subaction is called the *handler action*. The handler action gives a clean separation of the calling and called guardians and ensures that each individual action runs at just one guardian. It avoids anomalies such as an action that commits at one guardian and aborts at another. It allows the handler to commit or abort unilaterally, without concern about what the calling guardian does, and similarly for the caller.

A distributed program in Argus consists of a collection of guardians that may be dispersed geographically over the nodes of the network. A computation starts up as a topaction at some guardian and spreads to other guardians by means of handler calls. Executing a handler call might spawn further calls and might cause some objects to be modified. When the topaction commits, modifications made to stable objects by the topaction or its descendants at all the guardians it visited must be written to stable storage. To ensure that committing is atomic, we use the standard two-phase commit protocol [Gray 78]. The guardian where the action started acts as the coordinator; guardians visited by descendants of the action are the participants. The protocol is used only for topaction commits; a subaction commit is processed locally at the guardian where the subaction is running.

3 Viewstamped Replication

The semantics of Argus makes it easy for programmers to construct services that are highly reliable, that is, with high probability, the service does not lose information entrusted to it. But it provides no special support for availability. Instead, the Argus programmer must implement availability explicitly, by replicating important information at several guardians and implementing operations so that they run at enough places. Even though the semantics of Argus is helpful in implementing replication algorithms (operations can run within transactions), implementing availability can nevertheless be a difficult job. Our replication method addresses this difficulty by providing availability automatically without complicating the programming model. Programmers continue to write Argus programs as before, and the services implemented by the resulting programs are highly available.

This chapter provides an overview of our replication method. Section 3.1 describes the new primary copy method, which we call *viewstamped replication*. We state what the method achieves and give a sketch of its implementation. There are two parts to the method: section 3.2 describes the view management algorithm, which, when invoked, automatically reorganizes the replicas of a service to keep it available in spite of failures; section 3.3 describes transaction processing, which ensures that a topaction knows whether it can continue execution and eventually commit in spite of node and communication link failures. Finally, section 3.4 discusses the correctness conditions that must be satisfied.

3.1 Overview of the New Method

Our method works as follows. We replicate each individual guardian to obtain a *guardian group*. Each group consists of several instances called *cohorts*. These cohorts behave as a single, logical entity. Each cohort has a unique name called a guardian id (or *gid*, for short); the group as a whole bears a unique *groupid*. The set of cohorts is the group's *configuration*. Each cohort knows the groupid and the configuration to which it belongs. We assume that the configuration never changes, that is, the set of cohorts is fixed when the configuration is defined at guardian creation time; we briefly discuss this issue in Chapter 7.

A distinguished cohort is designated as the *primary*; it executes handler calls and participates in two-phase commit. The remaining cohorts are called *backups*; these are passive and receive state information from the primary. If a cohort fails or is partitioned, the remaining cohorts are reorganized and a new primary is selected, should the old one become inaccessible.

For the sake of illustration, let us look at a simple system consisting of a server and a client making requests to the server. The system is illustrated in Figure 3.1. The server is implemented as a guardian group bearing groupid G. In this particular example, there are five cohorts, *a*, *b*, *c*, *d*, and *e*. Cohort *a* is the primary P1. The remaining cohorts *b*, *c*, *d*, and *e* are backups. In the figure, the large circle enclosing the smaller ones represents the guardian group G and is meant to suggest that logically the cohorts are a single entity, even though physically they may be dispersed throughout a network.

When there are no failures, the client C communicates only with the primary cohort *a* of server G. The request is processed at *a* and the primary replies to the client. In the background, the primary communicates with backups, sending checkpoint information about the completed request.

Now suppose that a failure causes cohorts *b*, *c*, *d*, *e* to stop hearing from their primary; whether there is a network partition or a node crash is irrelevant. This situation is shown in Figure 3.2. In response to such a failure, the server reorganizes itself automatically to

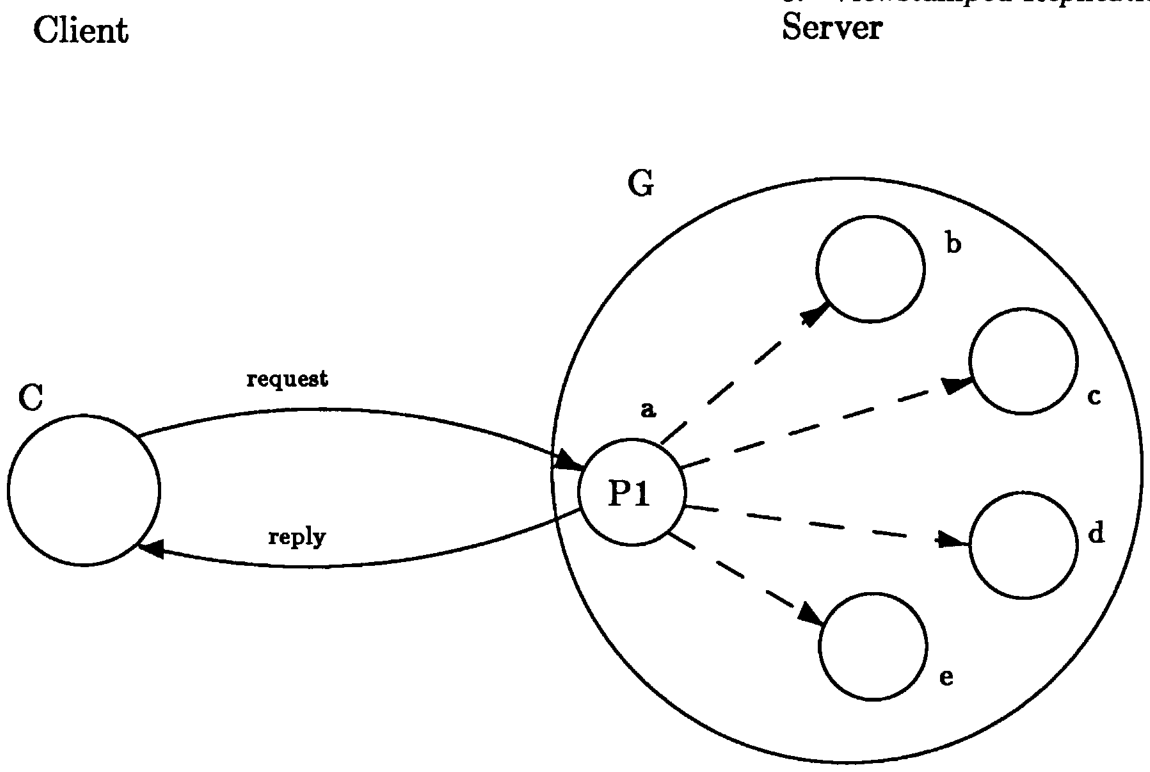

Figure 3.1: Normal operation. Client C sends requests to server G and receives replies. The dashed lines emanating from the primary cohort represent information flowing to the backups in background mode.

keep the service available. We call this reorganization a *view change*, and the algorithm that carries it out, the *view management algorithm.* In this example, the algorithm excludes cohort *a* and chooses a cohort, say *b*, as the new primary P2. Any new requests from the client will be sent to the new primary. A new primary can be chosen from the remaining backups as long as the backups and new primary together constitute at least a simple *majority* of all cohorts in the guardian configuration.

At this point an interesting question is whether actions can commit in spite of these view changes.

Consider the following scenario. Suppose topaction T1 made handler calls to server G before the view change. Now T1 attempts to commit; its coordinator sends out *prepare* messages to server G. When the new primary P2 (cohort *b*) receives the message, how does it know what to do with it? Can it prepare successfully? If P2 could somehow know whether the effects of T1 made at P1 propagated to the backups before the view change

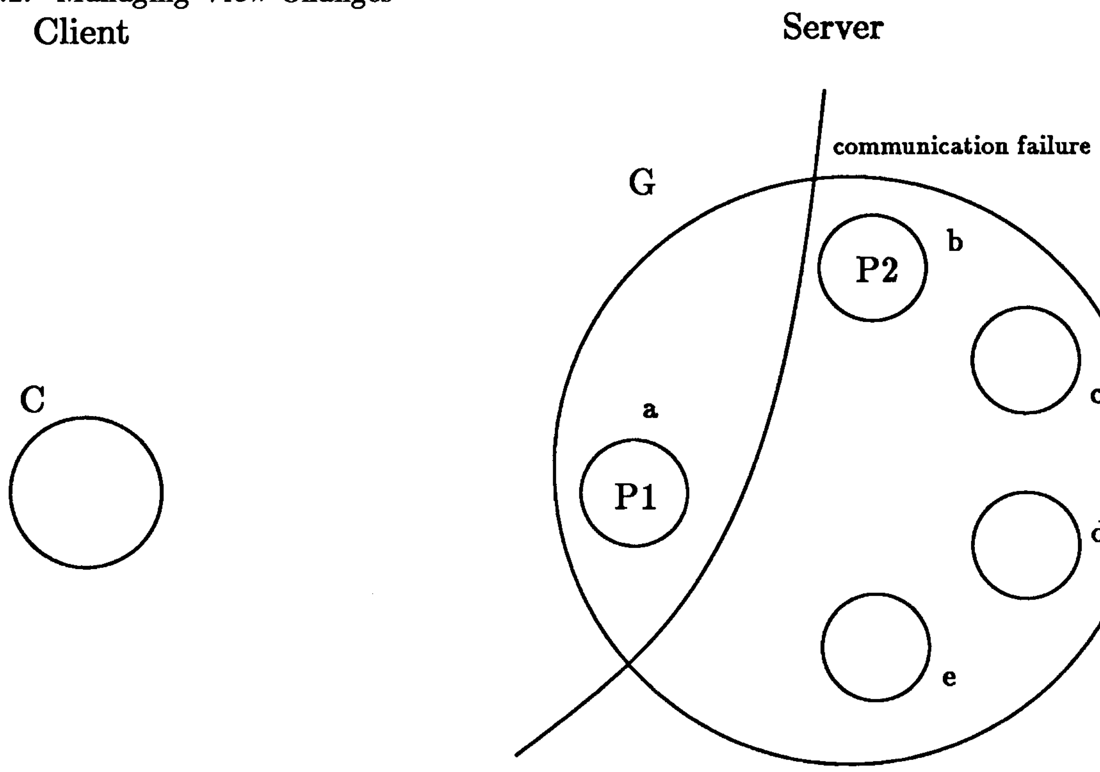

Figure 3.2: Client and server in the presence of communication failures.

occurred, then it could decide whether to accept the prepare.

Our replication method solves this problem, allowing the primary to determine inexpensively whether it can commit in spite of view changes. We have developed some additional mechanism to make it possible for a topaction to continue execution without aborting unless absolutely necessary. Our new method captures the notion of what cohorts "know" and furthermore, whether they know "enough" to permit a transaction to commit.

Our method consists of two parts: view changes and running transactions. These are discussed further below.

3.2 Managing View Changes

Over time, nodes and communication links may fail and recover arbitrarily, changing the topology of the network. These failures obviously affect the cohorts making up guardian groups. To mask these failures automatically and efficiently, and to preserve the single-image appearance of a guardian group, we introduce the notion of a *view*. Intuitively, a view reflects the changing communication capability among members of a configuration. For a guardian group, a view consists of a primary and some backups, together with who the primary is, and contains at least a majority of cohorts of the configuration. Each view is named by a unique identifier called a *viewid*; we guarantee that viewids are totally ordered.

For example, Figure 3.2 shows a new view $\{b : c\ d\ e\}$ containing cohorts b, c, d, and e in which b is the new primary P2; the old view was $\{a : b\ c\ d\ e\}$ with primary P1. In either case, the configuration is still $\{a\ b\ c\ d\ e\}$. Note that a view is always a subset of the guardian group's configuration.

In response to changes in communication capability in a view, the cohorts switch to a new view by executing the view management algorithm; our algorithm is a simplification and modification of the original virtual partitions protocol proposed by El Abbadi, Skeen, and Cristian [El Abbadi 85]. The view management algorithm tries to assemble a new view containing at least a majority of cohorts in the configuration; otherwise, cohorts remain in their old views. As part of the view change, the algorithm generates a new viewid.

In Figure 3.3, we illustrate what the view management algorithm achieves. Guardian group G has configuration $\{a\ b\ c\ d\ e\}$, and suppose its initial view $v1$ is $\{a : b\ c\ d\ e\}$, where a is the primary. Now suppose a communication failure makes it impossible for cohort a to talk to the others. When this failure is noticed, the system initiates a change in view. In our example, cohorts b, c, d, and e discover that they cannot talk to cohort a, their primary. G switches to new view $v2 = \{b :\ c\ d\ e\}$ where cohort b is the new primary P2. Notice that $v2$ consists of a majority of cohorts of the configuration.

In addition, a view change will be initiated at cohort a, but it will not succeed because there is no majority. In this case, cohort a continues to be in old view $v1$, but becomes *inactive*, which means that it refuses to process client requests.

As part of the view change, the algorithm selects an initial state of the guardian for the new view; all cohorts in the new view will be initialized with this state. The state consists of objects, together with their locks and tentative versions, if any. This initial state is obtained by finding a cohort that was in the previous view and had the most "recent" information; we use viewstamps to accomplish this, as explained in the next

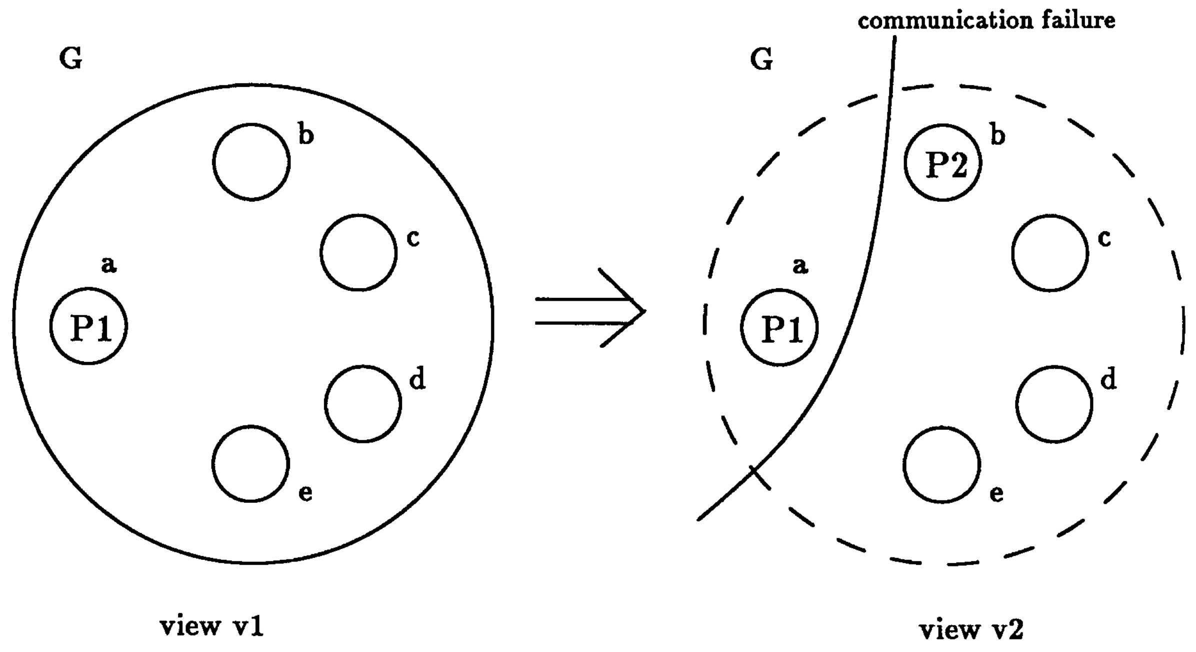

Figure 3.3: View changes. Guardian group G reacts to communication failures, such as a network partition, by changing views to $v2$ to exclude inaccessible cohort a.

section. We are guaranteed that at least one such cohort will exist because majorities must intersect.[1] That is, if the previous view contained a majority of cohorts, and the current view also consists of a majority, then both views must have at least one cohort in common that was in the last view and now is in the current view. Therefore, the new view starts out knowing what happened in the previous view. And since this intersection of majorities is true for all pairs of views, this cohort will also know what happened in all previous views, that is, everything that happened since creation of this guardian group.

3.3 Running Transactions

The view management algorithm guarantees that events that happened in the previous view are known in the next view. Transaction processing guarantees that needed events have "happened." Effects of a topaction that committed in that view or that prepared and did not abort will survive a view change; in addition, the effects of handler calls that committed locally in that view may survive. If the effects of its handler calls

[1]A sequence of failure events can invalidate this guarantee. We ignore this complication in our overview, but the situation is handled properly by our algorithm. The details are discussed in Chapter 5.

do survive a view change and are therefore known at the primary, we allow a topaction T to prepare. We use *timestamps* to determine whether the effects of a handler call are known.

3.3.1 Timestamps

The primary generates a new timestamp each time it needs to communicate information to its backups; each such occurrence is called an *event*. An *event record* identifies the type of the event, and contains the event's timestamp and other relevant information, which we defer discussing until later. An example of an event is the completion of the processing of a handler call or the commit of a topaction.

Timestamps are unique within a view and form a total order; they are easy to produce, for example, by incrementing a counter maintained by the primary. Each event is assigned a timestamp, and later events receive later timestamps. We require that the primary send event records to the backups in timestamp order.

Each backup receives event records from its primary in timestamp order and must process them in this order. Therefore, if a cohort knows about event x, it knows about all events that happened before x. More formally, we can state this prefix property as an invariant.

Invariant 1 *Within a view, for each event e with timestamp t, if a cohort knows e then it knows about all events e′ with timestamps* $t' < t$.

A primary maintains its timestamp to be the timestamp of the most recent event that has occurred. Each backup must record the timestamp for each event record it processes. Backups are inactive and shadow their respective primaries as far as what information they know; they only know what they have been told in checkpoint messages from their respective primaries, and are privy to nothing more. Therefore, their timestamps only reflect information received in checkpoint messages. Thus, the latest timestamp recorded at a cohort captures a portion of the past history of execution within the current view. These timestamps are an inexpensive way of determining what a cohort "knows."

3.3.2 Viewstamp history

To capture the fact that a timestamp is defined within a view, we also define *viewstamps*. A viewstamp is simply a timestamp paired with the viewid of the view in which the timestamp was generated. We refer to the parts of a viewstamp v as $v.id$ and $v.ts$.

Each cohort maintains a *viewstamp history* that represents the sequence of events from all views seen by this cohort. As noted above, the last timestamp within a view represents all past events for that view, so the history consists of a sequence of viewstamps, each with a different viewid.

For example, suppose we take a snapshot of a cohort at an instant in time, obtaining the following viewstamp history:

$$< v1,\ 10 >, < v2,\ 16 >, < v3,\ 4 >$$

From the first viewstamp $< v1,\ 10 >$, we can see that the state of the cohort reflects all events that occurred in view $v1$ up to timestamp 10: the cohort knows events $< v1,\ 1 >$ through $< v1,\ 10 >$ inclusive. Then the view changed to $v2$; the cohort now knows events $< v2,\ 1 >$ through $< v2,\ 16 >$ inclusive and those events are reflected in the cohort's state. Finally, in view $v3$, the cohort's state reflects only those events in that view whose timestamps are less than or equal to 4. Given the viewstamp history we know exactly what events are recorded at the cohort. In general, we guarantee that for each viewstamp v in the sequence, invariant 1 holds for each viewstamp, which we can also state as an invariant:

Invariant 2 *For each viewstamp vs $\in$ viewstamp_history, a cohort's state reflects all events e that occurred in the view of vs.id, such that $t \leq$ vs.ts, where t is the timestamp of e.*

Let us summarize.

1. *Views* automatically reorganize cohorts for availability.
2. *Timestamps* capture how much a cohort knows about the events that have happened in a view.
3. The *viewstamp history* captures how much a cohort knows about the events that happened in all views in the history.

In the next section, we show how this machinery allows topactions to determine whether they can commit.

3.3.3 Transaction processing

Running transactions requires the collaboration of both clients and servers. *Clients* create topactions, make handler calls to servers, and coordinate two-phase commit. *Servers* process handler calls, make further handler calls to other servers (thus acting as clients), and participate in two-phase commit.

Clients. When a client makes handler calls to servers, it includes the *action identifier* (or *aid*, for short) of the call action, the viewid that the client knows for the server in question, and other relevant information. The primary of the server processes the call and assigns the call a new viewstamp. This viewstamp flows back on the reply message to the caller, which remembers it in association with the groupid of the guardian group that ran the call. For example, in Figure 3.4, *aid* T1 and viewid *v*2 flow on the call message to server G. Viewstamp <v2, 6> flows back on the reply message.

At topaction commit, the client has a collection of viewstamps: there is at least one viewstamp associated with each server that participated in the topaction.[2] The client now acts as coordinator of two-phase commit. It determines who the participants are and sends *prepare* messages to their primaries; each prepare message contains the appropriate viewstamps for that server. The servers use these viewstamps to determine whether all effects of the topaction are known to them, as discussed further below.

Servers. When servers receive handler calls from clients, the calls are processed, viewstamps are assigned and returned, and, as noted above, are collected by clients. Later, the client sends the *prepare* message. Now consider the following situation. Each participant server receives a *prepare* message from the coordinator, containing the appropriate viewstamps. These viewstamps represent what *must be* known at the primary. The viewstamp history at the primary represents what *is* known. By simply checking that the received viewstamps are less than or equal to the associated viewstamps in the viewstamp history, we can determine whether servers remember enough to commit a topaction.

Figure 3.5 shows a *prepare* message arriving at the primary of participant server G, bearing some viewstamps for topaction T1 with respect to G. Recall that viewstamp

[2]If more than one call were made to the same server, there will be more than one viewstamp. We can optimize this situation by retaining the latest viewstamp for the set of viewstamps with the same viewid.

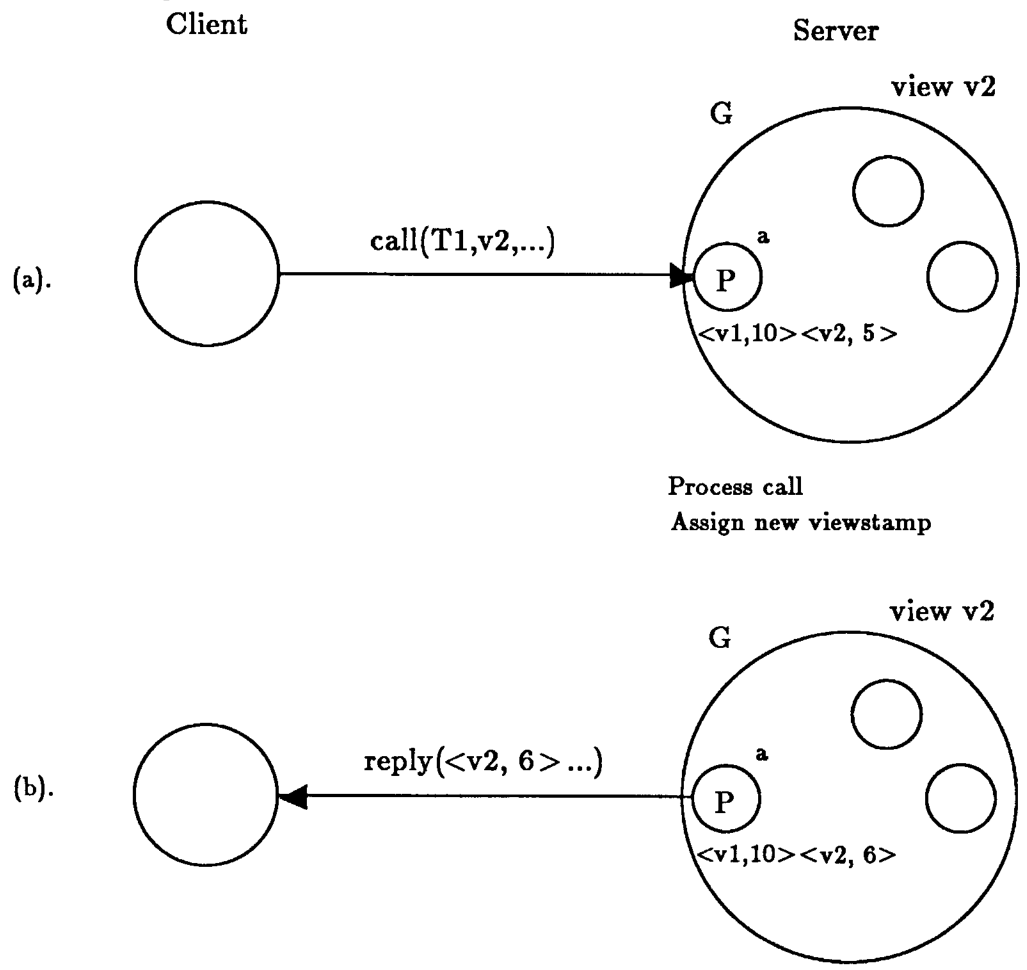

Figure 3.4: Handler calls. Client makes a call to a server, which processes it and replies to the caller. Viewids flow on call messages, and viewstamps flow back on reply messages.

$< v1, 10 >_{msg}$ (the subscript distinguishes viewstamps sent in the prepare message from those in the viewstamp history) means that the coordinator expects the cohort to know about all events in $v1$ up to timestamp 10; the history's viewstamp $< v1, 10 >_{hist}$ means that P really does know everything up to timestamp 10. No information is missing, and we are okay so far. Moving on to viewstamp $< v2,\ 6 >_{msg}$, we see that in view $v2$ the coordinator expects P to know at least timestamp 6. The history's viewstamp $< v2,\ 7 >_{hist}$ says that the cohort itself knows all events up to timestamp 7, inclusive. Since $6_{msg} \leq 7_{hist}$, P knows at least as much as the coordinator expects; in fact, it knows more, as indicated by event with timestamp 7. Since needed events for the preparing action are truly known at P, it is possible for topaction T1 to prepare at this participant.

Checking that needed events are known at the primary is not sufficient for preparing;

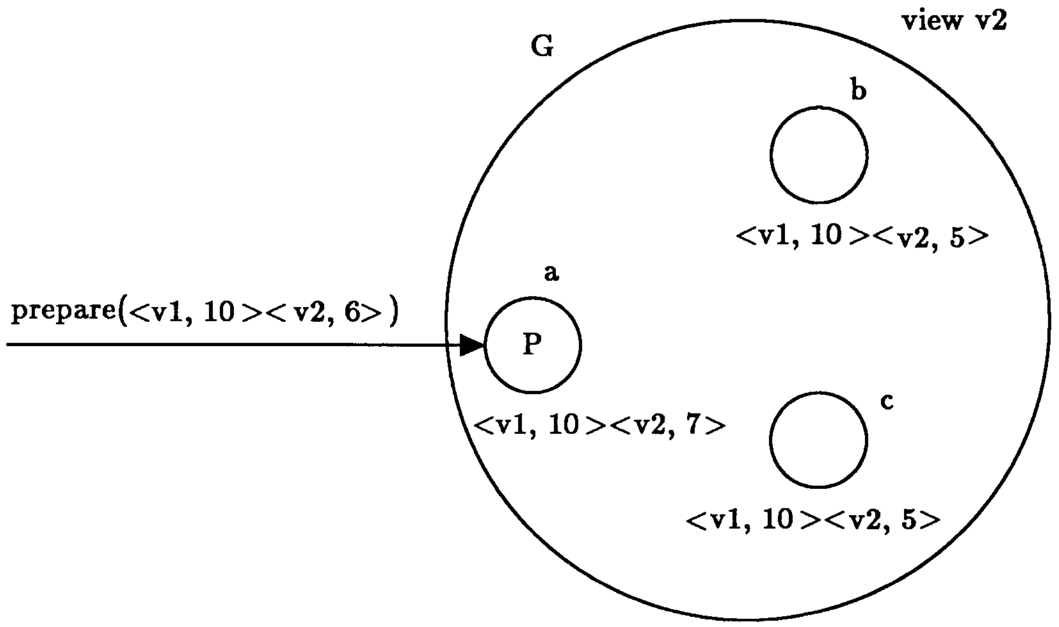

Figure 3.5: Can topaction T1 commit?

we must also ensure that the events are known at enough backups to survive a view change. The primary ensures this by forcing event records to the backups in the current view and waiting until a *sub-majority* know (a sub-majority is one less than a simple majority of the configuration) before responding *ok* to the coordinator. Other critical information is similarly forced to the backups, such as the committed record at the coordinator.

If any pairwise comparison fails, P is missing some events and topaction T1 *must* abort. To see this, instead of viewstamp $< v1,\ 10 >_{msg}$ suppose we had $< v1,\ 12 >_{msg}$. In this case, the test would fail (since 12 is greater than the timestamp in $< v1,\ 10 >_{hist}$) and the topaction must abort. This situation could arise because the view change from $v1$ to $v2$ happened before information about the handler call assigned viewstamp $< v1,\ 12 >$ was propagated to the backups.

3.4 Correctness

The correctness of our algorithm depends on the interaction of transaction processing and the view management algorithm. In this section, we discuss informally the conditions that must be met for correct operation. The conditions are the following:

1. In the absence of view changes, the system behaves as a non-replicated transaction system would.

2. A transaction can commit only if all of its events are known to at least a majority of cohorts.

3. The events known to a majority of cohorts of a configuration survive into all subsequent views.

Condition 1 is guaranteed by transaction processing. In a world in which failures never happen, our system behaves in the same fashion as a non-replicated transaction system.

Condition 2 is guaranteed by transaction processing. While a transaction is running at a server, event records are flowing to the backups from their primary. The transaction can commit at this server if all needed events are known at the primary as well as at at least a sub-majority of backups (at least a majority of cohorts in the configuration). To ensure that all needed events are known at the backups, the primary forces event records to the backups in the current view and waits for at least a sub-majority to respond.

Condition 3 is guaranteed by the view management algorithm. Since the previous view contained a majority of cohorts, and the new view also consists of a majority, both views must have at least one cohort in common. Furthermore, we select one of these cohorts that had the highest viewstamp, indicating the latest information. The new view starts out knowing what happened in the previous view. Thus, the effects of committed transactions survive in serialization order into all subsequent views.

4 Integrating Replication with a Transaction System

This chapter describes an implementation of the viewstamped replication method. In particular, we show how the implementation is integrated with the implementation of actions. We use the Argus implementation as a basis for this work.

Section 4.1 gives an overview of the Argus implementation of actions and the philosophy behind the design. Section 4.2 discusses the additional mechanisms that are needed to implement our replication method. Section 4.3 integrates our method with the implementation of actions. In particular, it describes transaction processing using nested actions. Section 4.4 describes what can be done in a system without nested actions to decrease the probability of topaction aborts when views change. Finally, section 4.5 discusses the performance of our algorithm.

4.1 Implementing Actions

In the Argus action system, the goal of avoiding unnecessary delays of user computations (while actions run and during two-phase commit) guided the implementation of atomic actions.

Delays assume either of two forms: extra communication and writes to stable storage. To avoid extra communication we piggyback information on messages that must be exchanged as a matter of course, and send other information in background mode. To minimize delay during writes to stable storage, several guardians participating in an action can write concurrently, and furthermore, can do some writing in background mode. We shall see shortly how the implementation has been tailored to accomplish these goals.

Some delays are unavoidable, however. For example, to commit a topaction it is necessary to communicate with all the guardians where descendants ran, and some writes to stable storage are needed (unless the action is read-only). As another example, lock conflicts between actions may introduce occasional delays.

In this section we describe the implementation of atomic actions in Argus. We describe the action tree, a useful way of visualizing the distributed state of a topaction at several guardians, explain the implementation of the two-phase commit protocol, and describe briefly how query messages hold the system together. More details are contained in the Argus implementation paper [Liskov 87].

4.1.1 Action trees

We can capture the distributed state of a topaction at a particular instant in time with an *action tree*. Each node of the tree is an action; the root is the topaction, and all the nodes descending from it are descendant subactions of the topaction. In this dissertation we will use the standard terms, *ancestor*, *sibling*, and *descendant*, to refer to the relationships among actions in the tree.

We label the root of the tree by the topaction's action identifier, or *aid* for short; the interior nodes are labeled by the *aids* of the descendant subactions of the topaction. Each node of the tree contains information about the status of its action, that is, whether it is *active*, *committed*, or *aborted*. For example, Figure 4.1 shows the status of topaction A and all of its descendants. A is active at guardian G (thus, A@G), A.1 and A.2.3 aborted, and the rest of the actions committed. For simplicity, we show only handler actions for handler calls. Recall that each handler call creates a *call action* at the calling guardian, and a *handler action* at the called guardian. All actions in the figure except A are handler actions, and each has as its parent a call action that is not shown in the figure. The notation A.1 means that A.1 is the first descendant subaction of action A; similarly for A.2. A.1 and A.2 may be subactions that ran either sequentially (A.1 then A.2) or concurrently.

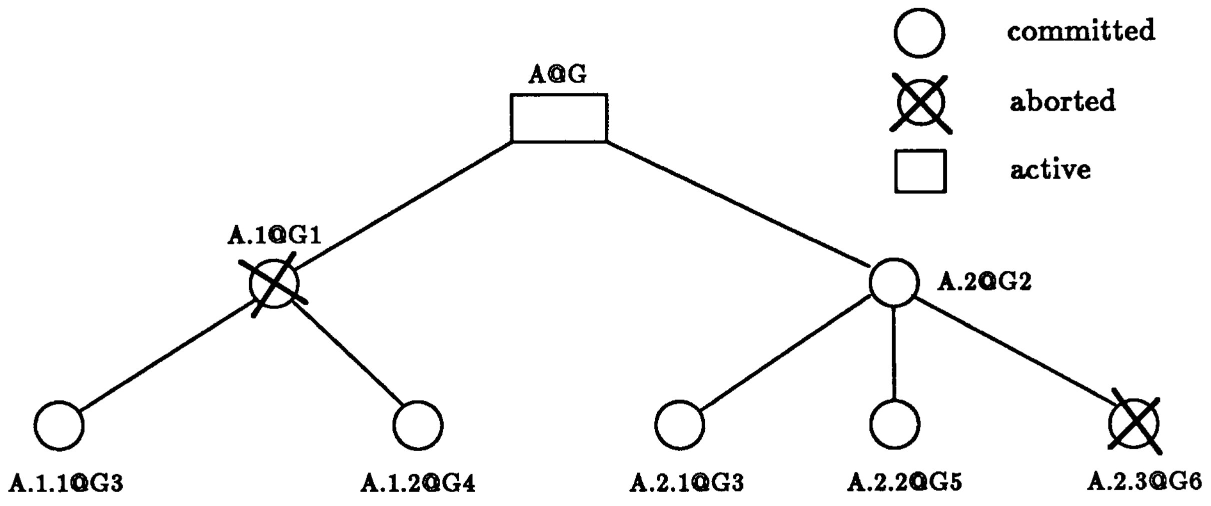

Figure 4.1: Action tree, showing the location and status of topaction A and its descendants.

We encode the structure of this abstract action tree in the *aids* themselves. *Aids* have the following properties:

1. An *aid* is globally unique.
2. An *aid* contains the identifier of the guardian where the action is executing.
3. An *aid* contains the *aids* of all ancestors of its action.
4. Given two *aids*, it is possible to tell whether one is an ancestor of the other.

To make the *aid* unique, each subaction appends the *gid* of the guardian at which it runs to a locally unique identifier, resulting in a pair <uid, gid> that is guaranteed to be globally unique. When a new handler subaction is created, a new *aid* is generated for it and is concatenated to the *aid* of the call action. Thus, an *aid* is a concatenation of <uid, gid> pairs.

A.2, A.2.1, and A.2.2 have *committed to the top*, which means that each has committed and so did all of its ancestors up to, but not including, the topaction A. A.1.1 and A.1.2 did not commit to the top, since A.1 aborted. If A commits at this point, all modifications to atomic objects made by A and all descendants that committed to the top must be installed and written to stable storage.

4.1.2 Implementing two-phase commit

When a topaction A attempts to commit, the system initiates a two-phase commit protocol at A's guardian. This guardian acts as the *coordinator* of the commit protocol, and communicates with the *participants*, which are the guardians where the descendants of A that committed to the top ran. If A commits in the example above, the participant guardians are G2, G3, and G5; it is not necessary to communicate with G1 or G6 because only aborted descendants of A ran there, and it is not necessary to communicate with G4 because the abort of A.1 undoes the effects of A.1.2. Note that information about the action tree must be communicated to the participants. For example, at G3 it must be possible to deduce that A.2.1's changes should be written to stable storage but not A.1.1's.

We use an algorithm in which the coordinator communicates with the participants directly in the first phase. To permit this, needed information is collected as handler calls commit at guardians and is passed up the tree to the next, higher level, when calls return. The collected information at the lower levels is merged with more information at the higher levels, until we reach the root, which will then know about all guardians that participated in the action. The coordinator can then communicate with all participants directly, passing them action tree information. This approach is consistent with our goal of limiting delay.[1]

The information collected *is* the action tree, but we keep it in a compressed form. On the assumption that commits are far more frequent than aborts, an *aborts_set* is maintained consisting of the highest aborted descendants, that is, aborted descendants for which no ancestor has aborted. Subactions that abort locally and have no non-local committed descendants are not remembered in the aborts_set, since all effects of such an action can be undone immediately without remote communication. In addition, a *parts_set* is maintained, containing the guardians of all committed descendants that are not descendants of the actions in the aborts_set; these guardians will become participants if the subactions eventually commit to the top. For example, just before action A in Figure 4.1 commits, we have parts_set = {G2,G3,G5} and aborts_set = {A.1}.

To commit a topaction, we need to know what objects it (or its descendants) used,

[1]An alternative approach in which two-phase commit messages fan out downward, level by level according to the action tree, may entail considerable delay for deep trees.

and, if it modified an object, we need to know the new version. This information is kept at the guardians where the objects reside. When an action is running, its guardian remembers all local atomic objects read or modified by it. When a subaction commits locally, this information is simply merged into the parent's information. When a subaction aborts, object information is discarded. When a handler commits, its guardian remembers its obj_set in a local, volatile data structure called *committed*. The *obj_set* records all the local atomic objects on which the subaction holds read or write locks.

Two-phase commit occurs as follows. The coordinator sends prepare messages, including the aborts_set, to the participants in the parts_set. The participants use the aborts_set to abort actions, if necessary, such as A.1.1. They use local information in *committed* to write the necessary information to stable storage, and then reply *ok* after the prepare record is written. If all say *ok*, the coordinator writes a committing record to stable storage and then enters phase two, sending commit messages to all participants. The participants use information in *committed* to install new versions, write a commit record to stable storage, and then reply *done*. When all reply, the protocol is over.

The information in the coordinator's parts_set and aborts_set plus the information in the participants' *committed* structure are sufficient to commit the topaction properly, *provided that no guardians crashed*. If a guardian crashes after running some handler calls that are subactions of topaction A, and then runs more handler calls that are subactions after it recovers, only the latter calls will be listed in *committed*. If a handler call that ran before the crash committed to the top, its versions should be written to stable storage. Since the versions were lost in the crash, the guardian ought to refuse to prepare. But given the information discussed so far, it cannot know this.

A guardian can recognize such problems by using a *crashcount*. Each guardian maintains a stable crashcount that records the number of times it has recovered from crashes. Whenever a handler commits, the current crashcount of its guardian is sent in the reply message. If another handler call is made to the guardian, the crashcount of the previous call is sent in the call message; if this number is less than the current crashcount, the call is rejected and the topaction aborts. Similarly, the crashcount is sent in the prepare message; if it is too small, the prepare is rejected.

4.1.3 Queries

Consistent with the goal of avoiding unnecessary delay is our decision that information about aborts and commits of subactions is not guaranteed to be propagated. For example, suppose a handler subaction aborted. We could immediately notify guardians where descendant actions ran and committed about the abort, but this would delay sending the reply message to the caller. Instead, we reply to the caller immediately and then communicate later at a convenient time with the guardians of descendant subactions. Furthermore, we do not try very hard to communicate, so there is no guarantee that the message will arrive. To mask lost messages, if some guardian needs to know what happened and did not receive a message, it can send a *query* message to the appropriate party. For example, if the coordinator aborts the topaction after phase one, it sends *abort* messages to the participants, but these messages may be lost. If an *abort* message fails to arrive at a participant that prepared in phase one, all is not lost, however, because that participant can query the coordinator to find out the fate of the action.

When a subaction commits, information about the action is stored locally and passed up the action tree, but never down. So if a concurrent or sequential relative of that committed subaction wants to obtain locks, the guardian must send query messages.

In general, query messages eliminate the need for reliable communication, placing the responsibility for making sure information is communicated on the guardian that needs to know what happened. Other guardians are thus relieved of the responsibility for delivering messages.

4.2 Implementing Viewstamped Replication

In this section, we describe the additional mechanisms needed to implement our replication method.

The new mechanism uses a buffer to communicate information from the primary to the backups, as described in the next section. Usually this information is sent in background, without the primary having to wait. Sometimes, however, the primary must wait until information has been recorded at enough backups. These times correspond to times in a conventional system when it is necessary to wait for information to be stored on stable storage. In effect, our system uses the backups as a replacement for stable storage.

4.2.1 Communication buffer

Recall that event records flow from the primary to its backups in timestamp order via checkpoint messages. In particular, instead of checkpointing event records directly to the backups, the primary maintains a communication *buffer* (similar to a fifo queue) to which it writes the event records. The primaries of both clients and servers make use of their own buffers to communicate information to their backups. The buffer provides the following operations:

1. *create().* Creates a new, empty buffer.
2. *add(e: event_record) returns (viewstamp).* This operation advances the timestamp of the current view, stamps the event record with the new timestamp, updates the viewstamp history, and appends the entry to the buffer. These four steps must be done atomically. It returns a viewstamp consisting of the current viewid and the timestamp created for this record.
3. *force-to(vs: viewstamp).* The operation returns immediately if *vs* is not a viewstamp for the current view. Otherwise, it delays the primary until a sub-majority of backups have received all event records in the current view with timestamps less than or equal to *vs.ts*.

In the *add* operation, the four steps must be done atomically. Recall that each cohort may have several processes running concurrently that could be calling the *add* operation. The implementation of *add* must serialize the use of the buffer and ensure that event records are recorded in the buffer in timestamp order.

Force-to delays its caller, but other work, including adding and forcing the buffer, can still go on at the cohort in other processes. If communication with some backups is impossible, the call of *force-to* will be abandoned, and the cohort will switch to running the view management algorithm.

The primary need not wait to hear from all backups in the current view. To reduce the period of waiting during a call to *force-to*, it is sufficient that the primary be delayed until only a *sub-majority* of backups knows all events whose timestamps are less than or equal to *vs.ts*. If a sub-majority of backups know about an event, a majority of cohorts in the configuration know about that event. Even though the primary is delayed until only a sub-majority know about events, the remaining backups in the view will eventually know about those same events unless a view change occurs in the meantime.

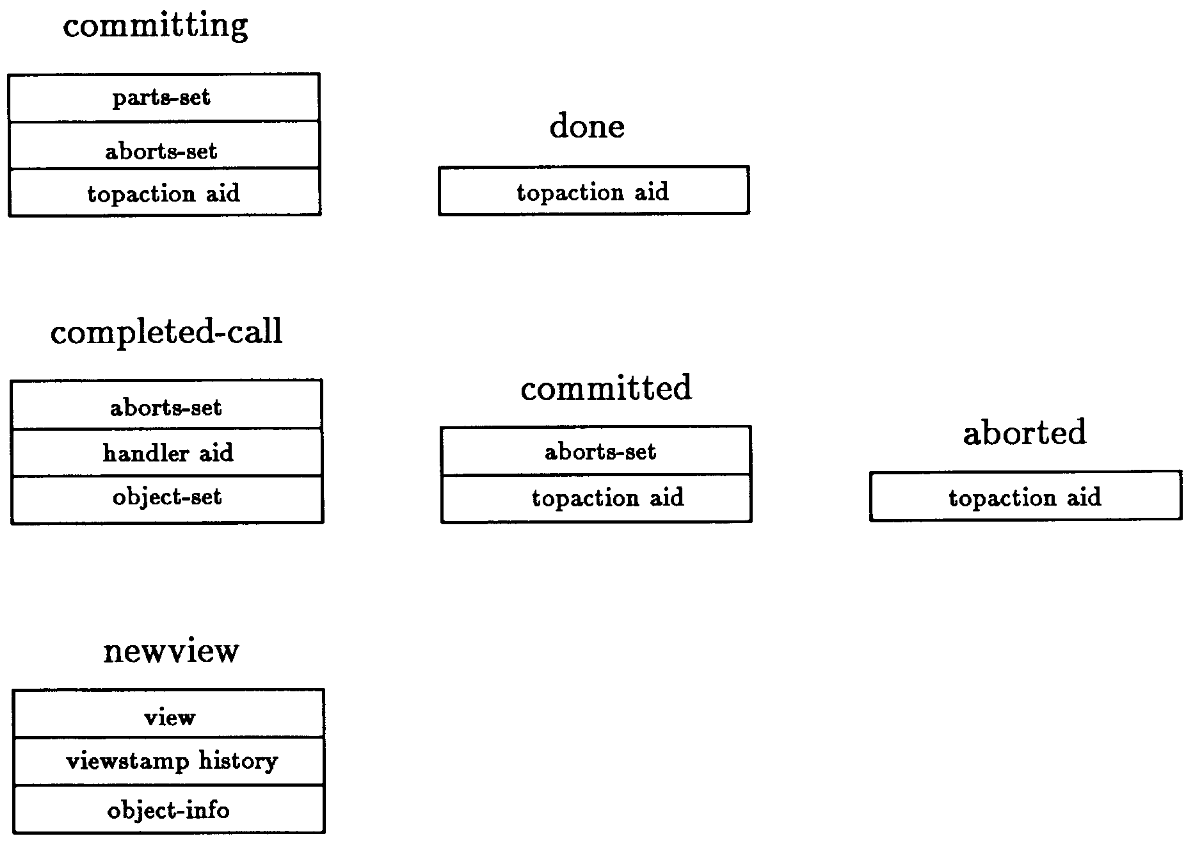

Figure 4.2: Format of event records.

Figure 4.2 shows the format of the six kinds of event records that can be written to the buffer. "Committing" and "done" are written by the coordinator of a committing action. "Completed-call", "committed", and "aborted" are written by the primaries of participants ("aborted" is also written by the primary coordinator). The "newview" event record is written by the new primary to its buffer after a view change. This record informs the backups in the new view of the view's membership (including who the new primary is), the viewstamp history, and the current state; it is discussed in Chapter 5.

The implementation of the buffer must deliver event records in timestamp order to guarantee the prefix property that any backup that knows about event e also knows about all events with timestamps less than e's. Furthermore, the buffer provides reliable delivery: any record added to the buffer will be delivered to all backups unless a failure occurs that will cause a view change. Acknowledgements from the backups are used at the primary to allow the *force-to* operation to complete. Also, event records that have arrived at all the backups can be removed from the primary's buffer.

4.2.2 Cohort State

The abstract state of a cohort is summarized in Figure 4.3. We show only the information relevant to transaction processing.

Each cohort has a *state*: it is "active" if it can participate in transaction processing, and otherwise it is involved in a view change. We say that a cohort is *active* if its state is "active"; otherwise, it is *inactive*. In particular, we will speak of active primaries of servers, that is, the active primaries receive client requests and process them.

Each *object* in the gstate has a unique name *uid* (relative to the group), a base version, and a set of lockers that identifies actions holding locks on the object, the kinds of locks held, and any tentative versions created for it. Figure 4.4 gives a pictorial representation of an atomic object as it might look in volatile memory.

Each cohort has a unique identity (*mygid*) and belongs to a *configuration*. The configuration is a set of identifiers that name the cohorts making up the group. Each configuration bears a globally unique groupid, which each cohort knows (*mygroupid*). *Timestamps* are generated only by the primary of a guardian group. Each cohort maintains a *viewstamp_history* that represents the sequence of view changes it has seen during its lifetime. Each member of the sequence is a viewstamp; for each viewstamp vs in the history, the cohort's state reflects each event in the view of $vs.id$ whose timestamp is less than or equal to $vs.ts$.

The *event-record* data type is a oneof, each of whose component tags corresponds to a different event record; the pictorial counterpart was shown in Figure 4.2.

4.2.3 Locating the primary cohort

How does system code at a guardian locate the primary of a guardian group? To find a server it has not used before, a cohort fetches the configuration from the location server and communicates with a majority of members of the configuration to determine the current primary and viewid. It stores this information in a local cache. It uses the cache when sending messages and updates it whenever it learns about view changes.

Although it would be possible to encode information about the configuration in the groupid, a better approach is to use a highly available location server that maps groupids

```
gstate: {object}                        % cohort state
state: status                           % cohort is active or doing a view change
mygid: int                              % unique name of this cohort
configuration: {int}                    % lists unique names of cohorts
mygroupid: int                          % unique name of guardian group
timestamp: int                          % timestamp generator
viewstamp_history: [viewstamp]          % indicates events known to cohorts
cur_viewid: viewid                      % identifier of the current view
buffer: [event-record]                  % communication buffer

where
   object =       <uid: int, base: T, lockers: {lock-info}>
   lock-info =    <locker: aid, info: oneof[read: null, write: T]>
   vid =          <cnt: int, gid: int>
   view =         <primary: int, backups: {int}>
   viewstamp =    <id: vid, ts: int>
   status =       oneof[active, view_manager, underling: null]
   event-record = oneof[committing: <parts_set: {groupid}, aborts_set: {aid},
                                    action: aid>
                       done: <topaction: aid>
                       completed-call: <aborts_set: {aid}, handler aid: aid,
                                        object-set: {object}>
                       committed: <aborts_set: {aid}, topaction: aid>
                       aborted: <topaction: aid>
                       newview: <cur_view: view, vs: viewstamp_history,
                                 object_info: {object}>
```

Figure 4.3: State of a cohort. {} denotes a set, [ ] denotes a sequence, oneof means a tagged union with component tags and types as indicated, and <> denotes a record, with component names and types as indicated.

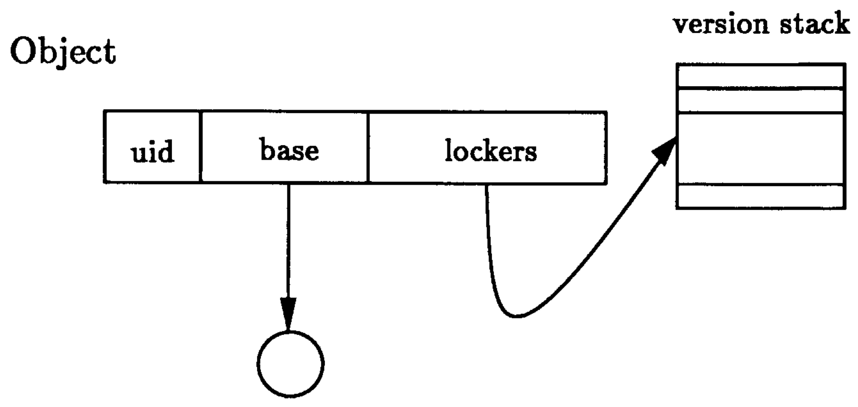

Figure 4.4: Representation of atomic objects.

to configurations. A location server can allow configurations to change; it also permits groupids to be smaller than would be the case if they contained configuration information within them. There are several ways of implementing such a server; see for example, Hwang's thesis [Hwang 87], forwarding addresses[Fowler 85], rendezvous [Mullender 85], and migratory objects [Henderson 82]. Note that the location server defines the limits of availability; no guardian group can be more available than it is.

All call messages contain the cached viewid for the server; replies indicate whether the view has changed. The system uses this information to keep the cache up to date.

4.3 Running Nested Transactions

Our replication system runs transactions in a manner similar to a system without replication. There are two main differences. First, we use viewstamps to determine whether a transaction can commit. Second, instead of writing information to stable storage during two-phase commit, the primary sends it to the backups using the communication buffer.

In this section, we describe an implementation for a replication system with nested actions. In particular, we explain what active primaries of clients and servers do, how cohorts process information they receive, and how query messages are used to compensate for lost information and to reduce the window of vulnerability in the two-phase commit protocol. We assume both clients and servers are replicated. In section 4.3.5, we discuss the usefulness of replicating clients and propose a way of making two-phase commit more robust for clients that are not replicated by using a replicated "coordinator-server."

4.3.1 Active primaries of replicated clients

Clients start topactions, make handler calls to servers, and coordinate two-phase commit. Figure 4.5 summarizes the processing that takes place at the primaries of clients.

Starting a topaction. To start a topaction, the primary of the client produces an *aid*. Since *aids* for topactions must be globally unique across view changes, we make the primary's *mygroupid* and *cur_viewid* part of the name, so we have <uid: int, groupid: groupid, viewid: vid>.

As in a non-replicated system, we must maintain an *aborts_set* and a *pset*, both initially empty. The aborts_set is the same as in our previous discussion of the implementation of actions. The *pset* is an analogue of the parts_set. Recall that each time the topaction makes a handler call, the server that processes the call assigns the call a new viewstamp upon completion of the call and returns the new viewstamp in the reply message. The coordinator collects these viewstamps in the *pset*, which is a set of <groupid: groupid, vs: viewstamp> pairs. Every committed handler call made by the topaction or its descendants has a pair in the pset showing the group where it ran and the viewstamp assigned to it. By the time the topaction is ready to commit, the pset represents the latest information known by the topaction about each of the participant guardian groups.

Making handler calls. To make a handler call, the system generates an *aid* for the call action. We produce the unique subaction *aid* by appending a <uid: int, groupid: int> pair to the parent's *aid*. Then the system looks up the primary and viewid for the group in its cache; if the server is not there the system communicates with the location server to fetch the configuration and then communicates with a majority of members of the configuration to find out the primary and current viewid; it stores this information in its cache. The call message is sent to the primary; the message contains the viewid from the cache, the call action *aid*, the handler id, and the arguments of the call.

There are two possible results of such a message. The first result is either a reply

Starting a topaction: Create the topaction *aid* and an empty pset and aborts_set.

Making a handler call:

1. Create the *aid* for the call subaction. Look up the server in the cache; if the server is not there then communicate with the location server to fetch the configuration and communicate with a majority of members to find out the primary and current viewid; store this information in the cache. Send the call message to the primary; the message contains the call *aid*, the cached viewid, handler id, and arguments.

2. If there is no reply or the reply indicates that the view has changed, abort the call action and add its *aid* to the parent's aborts_set. Then attempt to find out the new primary and viewid. If this succeeds, add the new information to the cache and go back to step 1. Otherwise, terminate the call and return to the user code with an exception indicating that the call cannot be completed right now.

3. If a successful reply message arrives, commit the call action, adding elements of the pset and aborts_set in the reply message to the parent's pset and aborts_set, respectively. Then return normally to the user code.

Coordinator for two-phase commit:

1. Determine who the participants are from the pset, and then send *prepare* messages containing the *aid*, aborts_set, and pset to the primaries of participants. Then act as a participant locally: release any read locks held by descendants of the topaction and discard any local locks and versions held by descendants of actions in the aborts_set.

2. Wait for responses.

   (a) If all participants agree to prepare, add a "committing" <parts_set, aborts_set, topaction *aid*> event record to the buffer; the parts_set is a list of non-read-only participants. Perform a *force-to(new_vs)*, where new_vs is the viewstamp returned by the *add* operation.

   If all participants are read-only, we are done.

   Send *commit* messages containing the aborts_set to the non-read-only participants; when all of them acknowledge the commit, add a "done" <topaction *aid*> event record to the buffer.

   (b) If any participant refuses to prepare, discard any local locks and versions held by the topaction's descendants. Add "aborted" <topaction *aid*> event record to the buffer and send *abort* messages to the participants.

   (c) If there is no answer after repeated tries, resend the *prepare* message to the new primary if there is one. Otherwise, abort the topaction.

Figure 4.5: Processing at active primaries of clients

indicating that the view has changed, or no reply at all (after a sufficient number of probes). In either case, the primary aborts the call action and adds the call action's *aid* to the parent's aborts_set. Then it tries to determine the current viewid and primary of the server. If it discovers new information, it updates its cache and tries the call again, generating a new call action with a different *aid*. Otherwise, it returns to the user code with a special exception indicating that the destination cohort is not responding. As in Argus, the intent of this exception is to inform the user code that an immediate retry is unlikely to succeed, but that a later attempt might succeed.

The second, and most likely outcome, is a reply message for the call. In this case, the call action commits. The reply message contains a pset and an aborts_set. The pset contains pairs for this call and any further handler calls made in processing it; the aborts_set contains *aids* of aborted descendants of the call action. The pairs in the reply's pset are added to the parent's pset, and the *aids* in the reply's aborts_set are added to the parent's aborts_set. In doing these additions, certain optimizations are possible to keep the sizes of the set small. For example, if some action A in the aborts_set is a descendant of a second action B, then only B need be retained since aborting it will undo the effects of all its descendants, including A. If there are two pairs in the pset for the same guardian group, and if the two viewstamps have the same viewid, then only the pair with the larger viewstamp need be retained, since any cohort that knows about the later viewstamp will also know about the earlier one.

Note that more than one pair for a server may appear in the pset due to view changes. We must keep pairs corresponding to different views because viewstamps are meaningful only for their own view; they imply nothing about events belonging to earlier views. For example, suppose action A makes a call A.1 to server G, and that this call runs and commits in view *v*1, and is assigned viewstamp <v1, 12>. This viewstamp is passed along in the reply message and is merged into the pset, resulting in pset {<G, <v1, 12>>}. G undergoes a view change, switching to new view *v*2. Then A makes call A.2 to G. A.2 commits and is assigned viewstamp <v2, 7>, which is sent in the reply message to the parent. If we tried to retain just the "later" viewstamp, G could agree to prepare by mistake, for example, if its viewstamp_history contained <v1, 6> since it would not know that it needs to know about <v1, 12>.

Coordinator for two-phase commit. When the topaction completes, the primary of the replicated client acts as the coordinator of two-phase commit.

The primary determines who the participants are from the pset. It sends *prepare* messages to the primaries of participants containing the committing topaction *aid*, the pset, and the aborts_set. The pset allows the participants to determine whether all effects of descendants of the top action at that participant are known. The aborts_set allows participants to undo effects of aborted descendants.

The primary of the coordinator then acts as a participant locally: it releases any read locks held by descendants of the topaction and discards any locks and versions for any action that is a descendant of some action in the aborts_set.[2]

If all participants agree to prepare, the coordinator adds a "committing" <parts_set, aborts_set, topaction *aid*> event record to its buffer, and forces the entire buffer to its backups. This ensures that the commit will be known across a view change of the coordinator. The parts_set lists only the participants where the topaction holds write locks, since only these must take part in phase two; the reply from the participant indicates whether or not it is read-only. Note that user code can continue running as soon as the "committing" record has been forced to the backups.

Then the coordinator sends commit messages containing the aborts_set. It continues to retransmit *commit* messages to a participant until it receives a *done* response. When it has received *done* messages from all participants, it adds a "done" <topaction *aid*> event record to its buffer.

We can optimize the coordinator protocol for read-only actions as follows: If the topaction is entirely read-only, there is no phase two. If the topaction is read-only at some participants, those participants need not participate in phase two; the coordinator sends *commit* messages only to the remaining participants (non-read-only) listed in the parts_set.

If any participant refuses to prepare (sends back a *refused* message), the coordinator sends *abort* messages to all participants. *Abort* messages are sent just once; they are not retransmitted and are not acknowledged. The primary of the coordinator adds an "aborted" <topaction *aid*> event record to its buffer. If some participant fails to respond

[2]The reason for this is that some descendant in the call chain might have made a handler call to the topaction's primary. But in reality these locks would have already been released.

after a reasonable attempt to communicate with it, the coordinator should try to find out if the view has changed and whether a new primary was chosen and then resend the *prepare* message to the new primary. Otherwise, abort the topaction.

The coordinator need not communicate with its backups upon aborting a topaction, that is, it need not add any information about the abort to its buffer. Suppose the coordinator's group later suffered a view change that led to a new primary. Since the coordinator's state is not checkpointed, any view change of the coordinator must cause any of the group's topactions to abort automatically. To avoid sizable amounts of lost work in the case of failures, the topaction should be structured to run as a series of short topactions [Gifford 85]. Short topactions have the added benefit of minimizing lock conflicts. Even though writing an "aborted" record to the coordinator's buffer is not necessary, we do it in order to speed up queries. For example, suppose a backup is cut off from the majority by a partition; if it received the "aborted" record, it can respond to a query about that topaction.

4.3.2 Active primaries of replicated servers

Servers process handler calls and act as participants in two-phase commit. They may also make further calls (thus behaving as clients) on other servers in the course of processing calls. Figure 4.6 summarizes the processing at the primary of a server.

The state of a cohort was described in Figure 4.3. Recall that the *gstate* contained the objects accessed by user actions. Actions that run at the primary acquire locks on objects in the gstate and create and modify tentative versions for them. (They may also create new objects.) In addition, while a handler call is running, it may use temporary objects that are discarded when it returns. To record the effects of a call, it is sufficient to record its effects on the objects in the gstate.

Processing a handler call. If the call message's viewid is not equal to *cur_viewid* the call is refused because in general it is not possible to know if this call is a duplicate. Otherwise, the primary creates an empty aborts_set and pset for the handler action and runs the call, possibly making further nested calls as described in section 4.3.1. When the call completes, it adds a "completed-call" <aborts_set, handler *aid*, object_set> event record to the buffer. Each member in the *object_set* identifies an atomic object that was read or written in the course of processing the call and indicates the type of lock obtained; if it is a write lock, the object contains the tentative version created for the subaction. Then the primary adds a pair <mygroupid, new_vs> for this call to the handler action's pset, where new_vs is the viewstamp returned by the buffer *add* operation, and returns the pset and aborts_set in the reply message. Finally, as in Argus, the primary records information about the handler action and the objects it used in *committed*.

If the handler action aborts, the pset in the reply message is empty. If the aborting handler action made no remote calls that may have committed at other guardian groups (that is, its pset and aborts_set are empty), the aborts_set in the reply is empty; otherwise, it contains the *aid* of the aborting handler action. Information about the action is discarded, and so are its locks and versions for local objects. Abort messages are sent in the background to all guardian groups in the pset.

When a primary of a server receives a call message for a handler call after the server had undergone a view change, there is in general no way for it to know whether the call had been run before the change. In particular, if the call *aid* in the message is not known to the primary, this might mean that this is a new call, or it might mean that the call ran before the view change or was running when the view change happened. In the first case, we need to redo the call; in the second case, we must *not* redo the call. To resolve this uncertainty, the server rejects the call. The client can then abort the call action, update its cache with the returned information, and retry in a new call action with a new call *aid* and new viewid. Aborting the old call action ensures that the nested call has no effect, and so there is no chance that the call will run more than once. Note that typically aborting the call is cheap because the call did not actually run. At worst we lose the work done by a subaction, but not the work done by the topaction.

Processing a prepare message. The primary can agree to prepare only if it knows about all handler calls it has done on behalf of the topaction. To determine if it knows, it uses its viewstamp_history and the pset in the *prepare* message: the pset tells it what "completed-call" events *must* be known, and the viewstamp_history tells it what events *are* known. If needed events are not known, it rejects the prepare and adds an "aborted" record for the topaction to the buffer. Otherwise, it forces the buffer enough to ensure that all "completed-call" records for the topaction are known at the backups and then

Processing a handler call:

1. If the call message contains the wrong viewid, send back a rejection message containing the new viewid and primary.

2. Create an empty pset and aborts_set. Then run the call. If it makes any nested calls, process them as described in Figure 4.5.

3. If the action commits, add a "completed-call" <aborts_set, handler *aid*, object_set> event record to the buffer; the object_set lists all objects used by the handler call, together with the type of lock acquired and the tentative versions if any. Add a <mygroupid, new_vs> pair to the pset, where new_vs is the viewstamp returned by the buffer *add* operation, and send a reply message containing the pset and aborts_set. Record the handler *aid* and obj_set in *committed.*

4. If the handler action aborts, send a reply message containing an empty pset. The aborts_set contains the handler action *aid* if there are committed, non-local descendants; otherwise it is empty.

Processing a prepare message:

1. If *compatible(pset, mygroupid, viewstamp_history)* is true perform a *force-to(vsmax(pset, mygroupid)).* Release read locks held by descendants of the topaction, discard locks and versions held by descendants of actions in the aborts_set, and reply *ok*. In the reply, indicate whether the topaction held only read locks at this participant. If the topaction is read-only, add a "committed" <aborts_set, topaction *aid*> event record to the buffer.

2. Otherwise, send a *refused* message to the coordinator refusing the prepare and abort the topaction; discard locks and versions held by its descendants and add an "aborted" <topaction *aid*> event record to the buffer.

Processing a commit message:

1. Add a "committed" <aborts_set, topaction *aid*> event record to the buffer. Perform a *force-to(new_vs)*, where new_vs is the viewstamp returned by the buffer *add* operation. Discard locks and versions held by descendants of actions in the aborts_set and then release locks and install versions held by descendants of the topaction. Finally, send a *done* message to the coordinator.

Processing an abort message:

1. Discard locks and versions held by descendants of the aborted action. If the aborted action is a topaction, add an "aborted" <topaction *aid*> event record to the buffer.

Figure 4.6: Processing at active primaries of servers.

sends a *prepared* message to the coordinator.

When the primary receives a *prepare* message, it compares the pset of the message with its viewstamp_history. We say the pset *is compatible with* the viewstamp_history of the primary if the primary knows about all handler calls done at its group on behalf of the topaction. More formally, we define *compatible* as a predicate on psets, as follows:

$$compatible(ps,\ g,\ vh) \equiv$$
$$\forall\ p \in ps\ (p.groupid = g \Rightarrow \exists\ v \in vh\ (p.vs.id = v.id \Rightarrow p.vs.ts \leq v.ts))$$

where ps is a pset, g is a groupid, and vh is the viewstamp_history.

We also need an auxiliary operation on psets. The $vsmax(ps,\ g)$ function, where ps is a pset and g is a groupid, returns the largest viewstamp associated with a handler call to the group (this is the viewstamp of the most recent "completed-call" event):

$$\begin{aligned} vsmax(ps,\ g) &\equiv max(\{p.vs \mid p \in ps\ \wedge\ p.groupid = g\}) \\ max(vs\_set) &= v_1 \in vs\_set\ s.t.\ \forall\, v_2 \in vs\_set \\ &\quad (v_2.id < v_1.id)\ \vee\ (v_2.id = v_1.id\ \wedge\ v_2.ts \leq v_1.ts) \end{aligned}$$

Note that *vsmax* is well-defined because there must be at least one pair p in the pset for this group.

If *compatible(pset, mygroupid, viewstamp_history)* is true, the primary forces the buffer by performing *force-to (vsmax(pset, mygroupid))*. Forcing ensures that the backups know about all events that preceded the reply of the last handler call to this group for the preparing action. The primary releases read locks held by descendants of the topaction that committed to the top. Locks and versions are discarded for any action that is a descendant of some action in the aborts_set. If the action now holds no locks, that is, all descendants that committed to the top only did reads, the primary adds a "committed" <aborts_set, topaction *aid*> event record to the buffer and sends an *ok-readonly* message back to the coordinator. Otherwise, the primary sends a *ok* message to the coordinator.

Even when an action has only read locks, we must force the "completed-call" records to the backups when preparing to ensure that read locks are held across a view change. A view change may have happened without this primary being aware of it, and there may be a new primary already processing user requests in the other view. Furthermore, the preparing action's read locks may not be known in the new view, so the new primary may allow other transactions to obtain conflicting locks. Forcing the buffer guarantees that the prepare can succeed only if the topaction's locks survived the view change. Without the force, the prepare could succeed at the old primary even if the locks did not survive.

If the pset in the message is not compatible with the viewstamp_history, the primary rejects the prepare: It adds an "aborted" <topaction *aid*> event record to the buffer, discards all locks and versions held for descendants of the topaction, and sends a *refused* message to the coordinator.

Processing a commit message. When a primary receives a *commit* message, it adds a "committed" <aborts_set, topaction *aid*> event record to the buffer and does a *force-to(new_vs)*, where new_vs is the viewstamp returned by the buffer *add* operation. It discards the locks and versions for descendants of actions in the aborts_set, installs new base versions and releases write locks, and sends a *done* message to the coordinator. If no *commit* message is received from the coordinator, the participant periodically retransmits its *ok* message.

We include the aborts_set in the *commit* message for the following reason. Having agreed to prepare, a participant may later undergo a view change that results in a new primary. Before committing the topaction, this new primary must discard the same locks and versions for descendants of actions in the aborts_set, as the old primary did. This new primary does not know what effects of the committing topaction to discard because it did not receive the *prepare* message that carried the aborts_set. Hence, the *commit* message must bear the the aborts_set.

Processing an abort message. When a primary receives an *abort* message, it releases locks and versions held by descendants of the action. If the aborted action is a topaction, it adds an "aborted" <topaction *aid*> event record to the buffer.

4.3.3 Other processing at cohorts

Cohorts that are not active primaries reject messages sent to them by other guardian groups, except for queries as discussed in the next section. The rejection message contains information about the current viewid and the identity of the primary if the cohort knows them (for example, if it is a backup in an active view).

Backups also receive messages containing information sent from the primary's communication buffer. Each such message contains the current viewid of its sender and a sequence of event records. Inactive backups discard such messages. An active backup discards any message whose viewid does not match the *cur_viewid*, and any duplicate entries in a message whose viewid does match. It processes accepted event records in timestamp order, updating its state accordingly, and sends an acknowledgement to the primary. It can simply store the entries, or it can perform them, for example, by setting locks and creating versions for a "completed-call" entry. Perhaps a good compromise is to store "completed-call" entries (as part of the gstate) until the "committed" or "aborted" entry for the call's topaction is received.

4.3.4 Queries

Recall from the earlier discussion in section 4.1.3 that the Argus implementation does not guarantee that all messages about transaction commits and aborts arrive where they might be needed. Query messages compensate for lost information. Under viewstamped replication, query messages are used in the same fashion. For example, the primary of the participant can send a query to the primary of the coordinator if the participant needs to know whether a topaction aborted. (The groupid in the *aid* of an action enables a cohort to find out to which guardian group it should direct its query.)

The only difference is that a query can be sent to other cohorts in a guardian group besides the primary; we allow any cohort to respond to a query whenever it knows the answer, in order to speed up the processing of queries. For example, queries are used in our system to reduce the window of vulnerability of two-phase commit. A cohort from the coordinator's group could be in an old view and may know that a topaction committed because this happened before the view change, and will respond *committed* to the participants. As another example, a cohort that is not a primary may know about the abort of a topaction because it received the "aborted" event record from the primary.

4.3.5 Non-replicated clients

This section discusses the usefulness of replicating clients, and proposes a way of making two-phase commit more robust for clients that are not replicated.

Why might replicated clients be useful? There are two kinds of clients. First, a client might actually be a server that does some tasks independently of handler calls. For example, a mail system might run this way, spooling mail for later delivery, and then delivering it in background mode. Replication is clearly useful for such a server.

Second, a client might act as a front-end module that interacts with a person at a console. The usefulness of replication is less clear in this instance. If the node on which the client is running fails (crashes or is partitioned), there may be no way some backup can take over communication with that particular console. Instead, the person may resort to some duplicate mechanism, such as another console or the telephone, that will put him in touch with a different front-end. These kinds of clients should probably run unreplicated.

Even when clients are unreplicated, however, it is useful to have a replicated coordinator, because this reduces the window of vulnerability during the two-phase commit protocol. When the coordinator is replicated, a participant will be able to determine the outcome of a topaction for which it is prepared if it can communicate with any cohort that knows the outcome. Furthermore, if the primary in the new view does not know about the commit, then the action can abort.

A replicated coordinator can be provided by means of a coordinator server. This is a guardian group that is used by an unreplicated client to start and end topactions. The client sends a *start-action* message to the coordinator server, which returns a topaction *aid* that the client then uses in all of its calls to regular servers. When the client is ready to commit, it sends a *ready-commit* message containing the *aid*, aborts_set, and pset to the server. The server then takes over coordinating the commit of the topaction. The primary of the server sends *prepare* messages containing the *aid*, aborts_set, and pset to the participants, and the protocol proceeds as described above. After the "committing" event record has been forced, it sends *commit* messages to the other participants and notifies the client that the topaction has committed. It also responds to queries about the outcome of the topaction; its groupid is part of the topaction's, so that participants know who it is. In answering a query about a topaction that appears to be still active it would check with the client but if no reply is forthcoming, it can abort the transaction unilaterally.

4.3.6 Discussion

There is a one-to-one correspondence between event records and information written to stable storage in a conventional system and therefore our system works because a conventional one does. The "completed-call" event records are equivalent to the data records that must be forced to stable storage before preparing, and the "committed" and "aborted" event records are the same as their stable storage counterparts.

The only difference is our treatment of topaction prepare. A peculiarity of our algorithm is that there is no analog of the prepare record that is written to stable storage in an ordinary two-phase commit. Prepares are not recorded in the communication buffer; they are only processed by waiting until a sub-majority of backups know about the effects of the prepared topaction. However, if a majority of the configuration (the primary plus the sub-majority of backups) know about the processing of the topaction, this is sufficient to ensure that the action's effects will survive subsequent view changes, and therefore the action will be able to commit.

In a conventional system, the prepare record serves two functions. First, it says who is prepared, so that we know to honor these promises. Second, it says who is *not* prepared; for these we can abort unilaterally.

In our method, the pset in the prepare message and the viewstamp_history tell the coordinator who is prepared; this information provides for the first function. The second function is much less important for our method. In a conventional system, recovering from a crash of a guardian can take a long time; a transaction that has not yet prepaed at this crashed guardian cannot commit and must abort. In our system view changes mask failures and are fast; the probability that an unprepared transaction will commit is high because the effects may have already been recorded at backups. Therefore, we try to commit here too. In a non-replicated system, if we want to try to commit after recovering, our method should be used. The added advantage is that preparing may sometimes be faster since information about the effects of handler calls can be recorded early. If all needed information is on stable storage at prepare time, we can avoid the synchronous delay.

4.4 Running Non-Nested Transactions

Nested actions are useful in our system because they are an economical way of masking the effects of view changes: We only abort the subaction, not the topaction. Furthermore, only when the view changes, do we need to abort and retry a subaction; thus, we do extra work only when the problem arises.

Without nested actions, however, aborting the call action is not an option. Instead, we must abort the entire topaction, in which case we are likely to lose work that has been done. In this section, we discuss how non-nested transactions run in a system that uses our replication method, and describe various ways of reducing the number of situations in which aborts occur.

To make a handler call, the system sends the call message to the primary. There are three possible results of such a message. The first, and most likely, is a reply message for the call; this result is handled in the same way as for nested actions.

The second possibility is no reply at all (after a sufficient number of probes by the system). In this case, we abort the topaction because we cannot know whether the call message would be a duplicate if we sent it to a new primary. The message might be a new one, or it might be a duplicate for a call that ran before the view change or was running when the view change happened. In the first case, we need to redo the call; in the second case, we must not redo it. To resolve this uncertainty, we abort the topaction.

The third possibility is a normal reply that also informs the caller that the view has changed. We update the cache with the new information. This possibility can arise if the server has undergone one or more view changes since the last handler call from the client but the primary remains the same throughout. In this case, the primary could maintain enough connection information to enable it to determine whether the call is a duplicate; if not, it can run the call and send back the special normal reply.

There are various ways of reducing the probablity that a topaction will abort. For example, we can keep client caches up-to-date by sending probes to find out the current view when a call is made to a group that has not been used for a while. Then when the call is made, it is highly likely to be made in the right view. Whether such an approach is practical depends on communication patterns. If the patterns are essentially static, that is, the client talks to a fixed set of servers and talks to them frequently, then the

	Non-replicated system	Viewstamped replication
RPCs		larger reply messages
prepare		larger prepare messages
best	force prepare record	no delay
worst	write changes; force prepare record to stable storage	force buffer
commit	force commit record to stable storage	add commit record to buffer and force

Figure 4.7: Performance.

information in the cache is likely to be up-to-date and probes will not be needed. If patterns are not static, however, then we may end up having to make two calls for each call, one to probe, and the second to make the call. Note that it will not work to combine the two calls; once the call message is sent, we must assume the worst (duplicate messages) if it is rejected. As another example, the primary could force a special "start_call" record to the backups before making any nested remote calls; in the absence of such a record, it would be safe for a primary to accept the call message even if the viewid in the call message is old. Neither of these techniques is satisfactory, however, since they delay normal processing.

4.5 Performance

In this section, we discuss how our method performs when running transactions and changing views.

Since operation calls execute only at the primary cohort and need not involve the backups at all, their performance is comparable to that in a non-replicated system. Messages under viewstamped replication are slightly larger because viewids must flow on every call message and possibly several viewstamps on every reply message.

In the best case, we expect that *prepare* messages will be processed entirely at the primary because the needed "completed-call" event records for handler calls of the preparing topaction will already be stored at a sub-majority of cohorts. In a non-replicated system, we incur a synchronous delay while the prepare record is written to stable storage. In the worst case, the primary must wait while the relevant part of the buffer is forced to the backups. Careful engineering is needed here to provide both speedy delivery and small numbers of messages. In a non-replicated system, we must write the modifications to stable storage, followed by the prepare record.

Committing a topaction requires forcing the "committing" event record to the coordinator's backups; the remainder of the protocol can run in background. For both preparing and committing, our method will be faster than using non-replicated clients and servers if communication is faster than writing to stable storage, which is often the case. Figure 4.7 summarizes this comparison.

5 View Management Algorithm

Transaction processing depends upon forcing information to the backups so that a majority of cohorts of a configuration know about particular events. The job of the view management algorithm is to ensure that events known to a majority of cohorts survive into subsequent views. The algorithm makes sure that every view contains at least a majority of cohorts and starts up the new view in the latest possible state.

If every view has at least a majority of cohorts, then it contains at least one cohort that knows about any event that was known to a majority of cohorts in the last view.[1] (The members in the last view have at least one cohort in common with the current view.) Thus, we need only make sure that the state of the new view includes what that cohort knows. We do this using viewstamps: the state of the cohort with the highest viewstamp for the previous view is used to initialize the state in the new view. This scheme works because event records are sent to the cohorts in timestamp order, and therefore a cohort with a later viewstamp for some view knows everything known to a cohort with an earlier viewstamp for that view.

This chapter describes the algorithm in detail. The next section gives an overview of the algorithm. Then we present the details. Next, we argue that the algorithm works by

[1]There is a situation involving a sequence of failure events that would invalidate this guarantee. We defer discussion of this point until section 5.2.

discussing its behavior in a variety of situations. Finally, we discuss the performance of the algorithm.

5.1 Overview of the Algorithm

Each cohort sends probe messages periodically to all other cohorts in its configuration, checking to see if the others are alive. If a cohort notices that it is not communicating with some other cohort in its view, or if it notices that it is communicating with a cohort that it could not communicate with previously, then it initiates a change in view. We call that cohort the *view manager*; the other cohorts are the *underlings*.

The algorithm operates in two phases. In phase one, the view manager invites all cohorts in the configuration to join the new view it will attempt to establish and waits for responses. It computes a new, unique viewid to name the new view and sends invitation messages to the other cohorts in its configuration. A cohort accepts the invitation only if it has not already received another invitation to join a higher-numbered view; each acceptance message contains the latest viewstamp known to that cohort, and also an indication of whether that cohort was the primary in the view of that viewstamp.

If less than a sub-majority accept the invitation, then the cohorts remain inactive and in their old views for a time, and then the algorithm is restarted.

If a sub-majority of cohorts accept the invitation, the view manager enters phase two to complete the view change. The cohort returning the largest viewstamp is selected as the new primary; the old primary of the view of that viewstamp is selected, if possible, to minimize disruption in the system. The manager sends a message to the new primary, notifying it about the new view; if the manager is itself the new primary, no message is sent.

If the cohort designated as the new primary does not refuse the message, it notifies the backups lazily about the new view by adding a special "newview" event record to the buffer. This event record contains the new view, the viewstamp history, and the primary's current state. The current state is a complete description of all object uids, base versions, lock information, and tentative versions if any. After adding the record, the cohort immediately becomes active and starts responding to handler calls. Eventually, during the course of normal transaction processing at the new primary, this "newview" record will reach all the backups in the new view. Each backup adopts the new view and updates its state to be the same as the primary's state contained in the event record.

5.2 The Full Algorithm

We now examine in some detail the algorithms to implement view management. Figure 5.1 illustrates the view management algorithm modeled as a finite state machine. A copy of the algorithm runs at each cohort. The state machine comprises three states: ACTIVE, VIEW_MANAGER, and UNDERLING.

A cohort is usually in the ACTIVE state. For the purposes of exposition in this chapter, there is little difference between the primaries and backups. It changes to the VIEW_MANAGER state when it detects any changes in the communication capability amongst cohorts. It makes a transition to the UNDERLING state upon accepting an invitation from another cohort.

In the VIEW_MANAGER state, a cohort changes state to UNDERLING if it accepts an invitation to join another cohort's view, or if it is not the new primary. If the cohort times out waiting for responses, it stays in the VIEW_MANAGER state. If it becomes the new primary, it makes a transition to ACTIVE; if some other cohort is chosen, it changes to the UNDERLING state.

In the UNDERLING state, if a cohort times out waiting for a *newview* message to arrive from the new primary, it switches to the VIEW_MANAGER state to start the algorithm all over again. When it receives either the *newview* message notifying it that it is the new primary or the buffer messages from the new primary, it switches to the ACTIVE state.

Figure 5.2 shows the program structure corresponding to the finite state machine. It is structured as an infinite loop. *State* is a tagged, discriminated union with component tags, each tag showing what state the machine is in. Associated with each tag is a program that is executed while the machine is in that state.

The remainder of this section shows the programs run in the various states. These programs communicate by using the **send** and **receive** statements.

The **send** m(args) **to** d statement sends message *m* to destination *d*. **Send** is unreliable; messages can be lost. A process receives messages by executing the **receive** statement; if there is more than one message waiting for it, one is selected non-

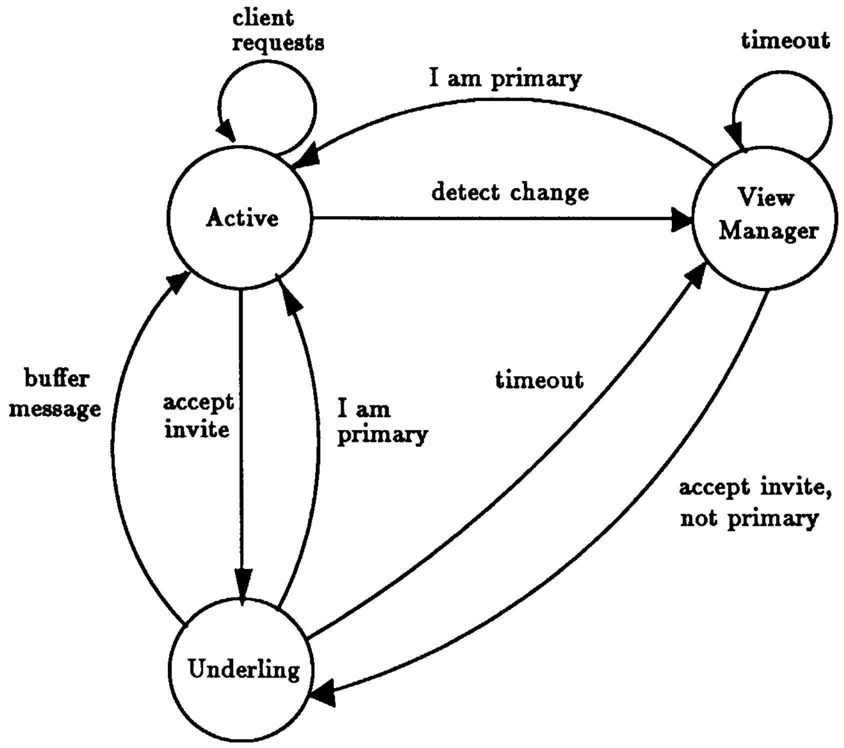

Figure 5.1: View management algorithm: Finite State Machine.

```
while true do
    tagcase state
        tag active: active()
        tag view_manager: viewmanager()
        tag underling: underling()
        end % tag
    end
```

Figure 5.2: State machine code.

deterministically. Then the arm corresponding to the name of that message is executed. The **receive** statement has an optional timeout parameter associated with it. For example, **receive within** t says that we wait for t time units for messages to arrive. If none arrive within the allotted time, the statement terminates with the *timeout* exception. Messages are sent both by other cohorts and by the system when probes indicate a change in communication capability.

5.2.1 Cohort state—view changes

Figure 5.3 shows the entire cohort state, including the variables associated with view management. In this section, we discuss this aspect of the cohort state.

Each cohort knows the view (*cur_view*) it is a member of as of the last view change and the identifier (*cur_viewid*) that uniquely names the view. *Max_viewid* is used during a view change to record the viewid of the view that is being formed and represents the highest-numbered viewid known by a cohort. *Cur_vs* is the current viewstamp and could be implemented as a pointer to the top of the viewstamp_history. *State* represents the three states of the finite state machine.

We assume that a cohort's state is stored in volatile memory and is lost in a node crash with the exception of the following four variables: *mygid*, *mygroupid*, *configuration*, and *cur_viewid*. We must remember these after a crash: *mygid* says who the cohort is, *mygroupid* says what group it belonged to, *configuration* says who the members of the configuration are, and *cur_viewid* says what the current viewid was before the crash.

We assume a small amount of stable storage to which every cohort has access. When the cohort is created as part of creation of the guardian group, the cohort stores *mygid*, *mygroupid*, *configuration*, and *cur_viewid*. Furthermore, we must write the *cur_viewid* as part of every view change. When a cohort recovers from a crash, it must initialize these variables by reading their values stored in stable storage. Then it initializes the variable *up_to_date* to be false. Finally, it starts up in the view_manager state.

5.2.2 The active state

Figure 5.4 shows a code fragment for the ACTIVE state.

In this state, a cohort could be either a primary or a backup. We do not show this part of processing, but just the processing pertaining to view management. There are

```
gstate: {object}                      % cohort state
state: status                         % cohort is active or doing view changes
mygid: int                            % unique name of this cohort
configuration: {int}                  % lists unique names of cohorts
mygroupid: int                        % unique name of guardian group
timestamp: int                        % timestamp generator
viewstamp_history: [viewstamp]        % indicates events known to cohorts
buffer: [event-record]                % communication buffer
cur_viewid: vid                       % identifier of current view
cur_view: view                        % the primary and backups
max_viewid: vid                       % highest viewid seen so far
cur_vs: viewstamp                     % current viewstamp
up_to_date: bool                      % true if gstate is meaningful

where
   object =        <uid: int, base: T, lockers: {lock-info}>
   lock-info =     <locker: aid, info: oneof[read: null, write: T]>
   vid =           <cnt: int, gid: int>
   view =          <primary: int, backups: {int}>
   viewstamp =     <id: vid, ts: int>
   status =        oneof[active, view_manager, underling: null]
   event-record =  oneof[committing: <parts-set: {groupid}, aborts-set: {aid},
                                       action: aid>
                         done: <topaction: aid>
                         completed-call: <aborts-set: {aid}, handler-aid: aid,
                                            object-set: {object}>
                         committed: <aborts-set: {aid}, topaction: aid>
                         aborted: <topaction: aid>
                         newview: <cur_view: view, vs: viewstamp_history,
                                   object_info: {object}>
```

Figure 5.3: Cohort state. {} denotes a set, oneof means a tagged union with component tags, and <> denotes a record, with component names and types as indicated.

```
active = proc()
   receive
      detect_change(new_viewid: vid):  % sent by probe monitor
         if new_viewid ≠ cur_viewid then return end % if
         state := view_manager
      invite(new_viewid: vid, g: int):
         if new_viewid ≤ max_viewid then return end % if
         max_viewid := new_viewid
         if up_to_date
            then send accept(mygid, max_viewid, cur_vs, is_primary?(mygid)) to g
            else send crash_accept(mygid, max_viewid, cur_viewid) to g
            end % if
         state := underling
      others:  % this is where transaction messages and queries are processed
               % and also (for backups) event records from the primary
      end % receive
   end
```

Figure 5.4: Active state.

two ways to change state: detecting changes in communication capability and accepting an invitation message.

We imagine that some process associated with each cohort monitors the "health" of other cohorts in the configuration. It probes other cohorts, checking to see if they are alive. If the set of responses differs from the current view, the process sends a detect_change message containing the *cur_viewid* to its cohort; otherwise, it starts another round of probing.

If a cohort receives a detect_change message from this probe monitor, it compares the incoming viewid in the message with the *cur_viewid*. If the viewids are the same, the cohort assumes the role as view manager and changes state to VIEW_MANAGER. Otherwise, the cohort ignores the message. We include the *cur_viewid* in the detect_change message to prevent unnecessary view changes. For example, if a cohort joined a view, became active, and then received an old detect_change message an unnecessary view change would be started. Why might a detect_change message be considered old? One reason is that the probe monitor was slow in delivering it. Another reason is that there could be several messages waiting and the process executing the **receive** statement selected another message, not the detect_change message.

A cohort g1 receiving an invitation message from some other cohort g2 accepts the invitation only if g2 knows a higher-numbered viewid. g1 records the *new_viewid* in *max_viewid*, sends an accept, and becomes an underling. *Max_viewid* represents the largest viewid seen so far by this cohort.

There are two kinds of acceptance messages, "normal" ones and "crashed" ones. If the cohort is up to date (that is, $up\_to\_date = true$), it sends a normal acceptance containing its identity, its *max_viewid*, its *cur_vs*, and an indication of whether it is the primary in the current view. *Is_primary?(mygid)* is a function that returns true if *mygid* is the primary in the view of the current viewstamp, or false otherwise. If the cohort is not up to date, it sends a "crash_accept" response; this response contains its *max_viewid* and its *cur_viewid*, and indicates that it has forgotten its gstate.

5.2.3 The view manager state

Figures 5.5 and 5.6 show the details of the algorithm run by the view manager. The view manager first discards the contents of its buffer. Then it computes a new, unique viewid composed of the successor of the largest sequence number in a viewid seen so far and its gid, *mygid*, and records it in *max_viewid*. It sends a message to all cohorts in the configuration (excluding itself), inviting them to join the view identified by *max_viewid*.

Responses are recorded in a *responses* data structure that maps gids to normal acceptances or crash acceptances. The cohort records a crash acceptance for itself if *up_to_date* is false; otherwise, it records a normal acceptance.

In the **receive** statement, the cohort waits on three kinds of messages: accept and crash_accept messages from cohorts that agree to accept the invitation and invitation messages from other cohorts that think they are view managers. The δ_1 parameter indicates that the cohort waits δ_1 time units to cover the time needed to transmit the invitation messages and the time for the acceptances to flow back to the sender.

Upon receiving a normal accept message, the cohort checks the viewid in the message against its *max_viewid*. If they are the same, then the cohort inserts a normal response in *responses*, keyed to the gid of the sending cohort. If the insert operation discovers that there is already an entry for g, it remaps g to the newly created response record. This handles the situation in which the view manager receives several messages from some underling. If the number of responses is equal to the number of cohorts in the configuration, the cohort has heard from everyone before the timeout expired and breaks

```
viewmanager = proc()
   buffer := buffer$new()
   max_viewid := vid${cnt: max_viewid.cnt + 1, gid: mygid}
   for c: int ∈ configuration − {mygid} do
      send invite(max_viewid, mygid) to c
      end % for
   response = oneof[normal: <cur_vs: viewstamp, prim?: bool>, crash: <cur_viewid: vid>]
   responses: map[gid,response] := create()
   resp: response
   if up_to_date
      then resp := make_normal(max_viewid, cur_vs, is_primary?(mygid))
      else resp := make_crash(cur_viewid)
   insert(responses, g, resp)

   while true do
      receive within $\delta_1$
         accept(g: int, new_viewid: vid, new_vs: viewstamp, prim?: bool):
            if new_viewid = max_viewid
               then insert(responses, g, make_normal(new_vs, prim?))
                    if size(responses) = |configuration| then break end % if
               end % if
         crash_accept(g: int, new_viewid, cur_viewid: vid):
            if new_viewid = max_viewid
               then insert(responses, g, make_crash(cur_viewid))
                    if size(responses) = |configuration| then break end % if
               end % if
         invite(new_viewid: vid, g: int):
            if new_viewid ≤ max_viewid then continue end % if
            max_viewid := new_viewid
            if up_to_date
               then send accept(mygid, max_viewid, cur_vs, is_primary?(mygid)) to g
               else send crash_accept(mygid, max_viewid, cur_viewid) to g
               end % if
            state := underling
            return
         others: % Queries are processed here
         end % receive
      end % while
      except when timeout: end % except
```

Figure 5.5: View manager state.

```
if ~maj(responses, configuration) then return end % if

new_primary: int := choose_primary(responses)   % Have majority
  except when no_view: % wait some time
                         return end % except
backups: {int} := all(responses) – {new_primary}
new_view: view := <new_primary, backups>
if new_primary ≠ mygid
  then send newview(new_view, max_viewid) to new_primary
       state := underling
  else if cur_view.primary ≠ new_primary then abort topactions end % if
     cur_viewid := max_viewid  % view manager is new primary
     cur_view := new_view
     timestamp := 0
     cur_vs := <cur_viewid, timestamp>
     append cur_vs to viewstamp_history
     add "newview" <cur_view, viewstamp_history, object_info> to buffer
     write cur_viewid to stable storage
     state := active
  end % if

end
```

Figure 5.6: View manager state (continued).

the loop. The view manager processes a crash_accept message from a cohort g like a normal accept, except that it inserts a special crash response record in the *responses* map, keyed to g. (In the code, a **break** statement causes an exit from the smallest containing loop.)

In an incoming invitation message if the *new_viewid* is greater than the *max_viewid*, the recipient records the *new_viewid* in *max_viewid*, sends either a normal accept message or crash_accept message to the manager, and changes state to UNDERLING as discussed below. Otherwise, it ignores the message because it comes from a cohort with a smaller viewid. (In the code, **continue** causes control to continue with the next iteration of the smallest containing loop.)

When the **receive** statement times out we need to know whether at least a majority has responded. Figure 5.6 shows the remainder of the view manager code. *Maj* is a predicate that returns true if the number of responses is at least a majority of cohorts of the configuration. If it returns false, then we wait for a while and then restart the algorithm in the VIEW_MANAGER state.

If we have at least a majority, the function *choose_primary* selects the cohort with the highest viewstamp as the new primary, using the old primary if possible; the view manager is itself selected if there is a tie and no old primary can be chosen (we do this to avoid extra messages). It may raise an exception *no_view* if a view cannot be formed; we defer discussion of the precise rules governing view formation. The function *all* returns the set of gids of all cohorts that responded; we then compute the backups by excluding the new_primary's gid from the set. The view manager assembles the new view consisting of the new primary and the backups. If the new primary is another cohort, the view manager sends that cohort a *newview* message containing *max_viewid* and the new view and enters the UNDERLING state to await information from the new primary.

If the new primary is the same as the view manager, then it does the following. It checks to see if it was the primary in its previous view; if not, it aborts any of its active topactions, that is, those for which it would be the coordinator. It records *max_viewid* in *cur_viewid* and also the new view in *cur_view*. It initializes *timestamp* to 0. It creates a new viewstamp <cur_viewid, timestamp>, denoted by *cur_vs*, and appends it to the viewstamp_history. It adds a "newview" <cur_view, viewstamp_history, object_info> event record to the buffer, where object_info contains the full gstate. This information can be sent in the background, but should be sent quickly to prevent another cohort from starting a view change. Finally, it writes the cur_viewid to stable storage. Then it enters the ACTIVE state. At this point in time, the view "takes effect," so to speak. Only the new primary knows which cohorts are in the view and what the new viewid is.

View formation can succeed only if two conditions are satisfied: at least a majority of cohorts must have accepted the invitation, and at least one of them must know all forced information from previous views. The latter condition may not be true if some acceptances are of the "crashed" variety. To see why forming a view is wrong if the latter condition is false, suppose a guardian group G consists of five cohorts g1, g2, g3, g4, g5 in view $v1$ = {g1: g2 g3 g4 g5}, where g1 is the primary. A partition separates g4 and g5 from the rest; they remain in the old view $v1$, while $v2$ = {g1: g2 g3} is formed from the others. Next, g1 crashes and recovers, losing its state. Then the old partition is repaired, making g4 and g5 accessible once again, but a new partition isolates g2 and g3. Cohorts g1, g4, and g5 are able to talk to one another. In the previous view $v2$, g1 knew about events that were forced to a majority of backups, but since it crashed, it now knows nothing at all. If g1, g4, and g5 were to form a new view, g1's state would be updated with that of g4's and g5's, which is out of date. This new view *v*3 would *not* contain what g1 knew, because g1 crashed.

To prevent this erroneous view formation, the last thing the new primary does during a view change is to record its *cur_viewid* on stable storage; the backups record the cur_viewid on stable storage while processing the newview event record. After a crashed cohort recovers, it sets *up_to_date* to false and starts up in the VIEW_MANAGER state. *Up_to_date* allows us to detect a situation like the one above and prevent view formation. In the example above, g1 has a larger *cur_viewid* than g4 or g5, but g1 is not up-to-date. Thus, we avoid forming the erroneous view, and instead, we wait a bit and then restart the algorithm in the VIEW_MANAGER state.

The correct rule for view formation is as follows: A majority of cohorts have accepted invitations and, in addition,

1. a majority of cohorts accepted normally, or
2. crash_viewid < max_viewstamp.id, or
3. crash_viewid = max_viewstamp.id and the primary of view max_viewstamp.id has indicated a normal acceptance of the invitation.

where crash_viewid represents the largest viewid returned in a "crashed" acceptance, and max_viewstamp represents the largest viewstamp returned in a "normal" acceptance. Condition (1) says we can ignore crashed acceptances if we have enough normal ones; condition (2) says we can ignore crashed acceptances if they are from old views; and condition (3) says we can ignore a crashed acceptance if we have information from the primary of its view, because the primary always knows at least as much as any backup.

5.2.4 The underling state

Figure 5.7 shows the code executed in the UNDERLING state. First, the cohort discards the buffer; as before, this has an effect only if it was the primary. Three things can cause a change in state: receiving invitation messages, newview messages, or buffer messages.

If the underling receives an invitation message from another cohort it accepts the invitation provided that the incoming viewid is greater than or equal to *max_viewid*;

otherwise, it ignores the invitation because it comes from a view manager that knows a smaller viewid. The *new_viewid* is recorded in *max_viewid.* If *up_to_date* is true, then it sends a normal acceptance message; otherwise, it sends a crash_accept message. It remains in the UNDERLING state.

If the underling receives a newview message, this means that the view manager whose view it agreed to join has appointed it the new primary. The message contains the new view *nv* and viewid *max_v* of the new view. The message is accepted only if the underling knows the same viewid; this ensures that since agreeing to join this view manager's view, the underling has not joined another, higher-numbered view. Also, if the cohort designated as the new primary crashed and lost its state before receiving the new view, it would be wrong for it to be the primary; in this case (*up_to_date* is false) the cohort discards the message. Otherwise, the underling records the *max_viewid* as the current viewid and so on, just as the view manager did when it appointed itself as the new primary.

The final possibility is that the underling receives a buffer message from the new primary. This message contains a sequence of event records and their timestamps. The cohort discards the message if its viewid is not equal to *max_viewid.* Otherwise, the cohort uses the first event record in the message (the "newview" record) to initialize the cohort state and writes cur_viewid to stable storage. Then, *up_to_date* is set to true, the other event records are processed, and the cohort becomes active.

The reason for the δ_2 timeout is the following. δ_2 covers the time for the underling to send an *accept* message to the view manager. If the view manager is not the new primary, it takes additional time for the view manager to notify the new primary, and it may take even longer for the new primary to begin informing its backups.

5.3 Why the Algorithm Works

We claim that our view management algorithm assembles a new view and that the new view is initialized with the latest state information. Each view intersects with the next view, so that each view contains at least one cohort that knows about previous events.

In this section, we argue that the view management algorithm is robust in the face

```
underling = proc()
  buffer := buffer$new()
  while true do
    receive within $\delta_2$
      invite(new_viewid: vid,  g: int):
        if new_viewid $\leq$ max_viewid then continue end %  if
        max_viewid := new_viewid
        if up_to_date
          then send accept(mygid, max_viewid, cur_vs, is_primary?(mygid)) to g
          else send crash_accept(mygid, max_viewid, cur_viewid) to g
          end %  if
        return

       newview(nv: view,  max_v: vid):
         if max_v $\neq$ max_viewid | ~uptodate then return end %  if
         cur_viewid := max_viewid
         cur_view := nv
         timestamp := 0
         cur_vs := <cur_viewid, timestamp>
         append cur_vs  to viewstamp_history
         add "newview" <cur_view, viewstamp_history, object_info> to buffer
         write cur_viewid to stable storage
         state := active
         return

       buffer(viewid: vid, msg: [<ts: timestamp, e: event_record>]):
         if viewid $\neq$ max_viewid then return end %  if
         for ts: timestamp, e: event_record in messages$elements(msg) do
           tagcase e
             tag newview(nv: view, hist: [viewstamp], objs: {object}):
                   viewstamp_history := hist
                   gstate := objs
                   cur_view := nv
                   cur_viewid := max_viewid
                   cur_vs := top of viewstamp_history
                   write cur_viewid to stable storage
                   up_to_date := true
             others: %  handle "regular" event records here
             end %  tagcase
           end %  for
         state := active
         return

        others:  %  Queries are processed here
        end %  receive
        except when timeout: state := view_manager
                                 return end %  except
     end %  while
  end
```

Figure 5.7: View underling state.

of simple failures, concurrent view managers, and two coexisting primaries.

5.3.1 The simple case

Figure 5.8 illustrates the operation of the algorithm. We assume for simplicity that following the initial failure, no additional failures occur; once cohort gl becomes inaccessible, it remains inaccessible for the duration of the algorithm. In this section, we describe how the algorithm operates in this simple case.

Figure 5.8(a) shows guardian group G consisting of five cohorts. gl is the primary, and the remainder are backups. For group G, viewid *v*1 identifies view {gl: g2 g3 g4 g5}. Each cohort, as usual, has its own viewstamp history. Cohorts send and receive probe messages.

A communication failure makes cohort gl inaccessible, as we can see in Figure 5.8(b), and g2, g3, g4, and g5 stop hearing from it. We suppose that g3 detects this change and galvanizes the algorithm into action.[2] g3 becomes the view manager and enters the first phase of the algorithm. It computes a new viewid <2, g3> by incrementing the first component of its *max_viewid* and concatenating its *my_gid*. This viewid is higher than anything g3 has seen. Next, it sends invitation messages containing the new viewid <2, g3> to other cohorts in the configuration (gl, g2, g4, and g5) and waits for responses.

In Figure 5.8(c) each cohort that received the invitation message sends back an acceptance message containing, among other things, its current viewstamp. To avoid cluttering the picture, we ignore the viewid and whether that cohort was the primary in the view of that viewstamp. No reply is forthcoming from gl since it is inaccessible. g3 collects the responses.

In phase two of the algorithm, illustrated in Figure 5.8(d), g3 considers the responses and arbitrarily selects g2 to be the new primary (all underlings have the same latest viewstamp). g3 assembles the new view {g2: g3 g4 g5}, identified by v2 = <2, g3>. Finally, g3 sends a *newview* message to g2 containing {g2: g3 g4 g5} and <2, g3>. If g3 were the new primary, no message would be sent. The algorithm is done.

We noted earlier that at this point in time the new view exists only after the new primary receives the newview message. The other cohorts do not yet know (with the

[2]More than one cohort may detect this change and trigger the algorithm, but we defer discussion of that possibility until the next subsection.

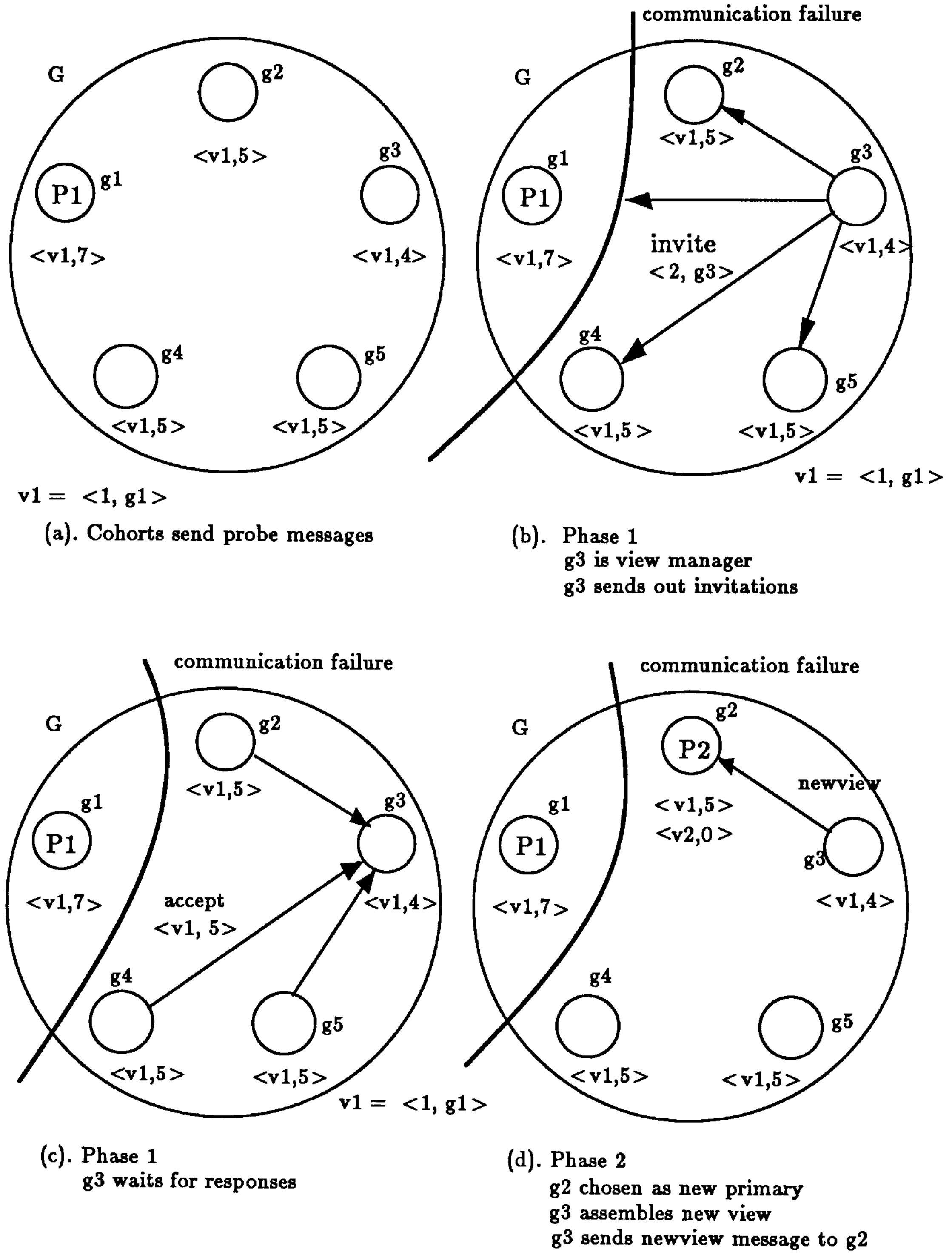

Figure 5.8: Simple case—Detecting a communication failure triggers the view management algorithm.

possible exception of the old view manager that formed the view) what the view actually consists of; they will discover that fact when they receive the "newview" event record from the new primary.

In the meantime, while all this is going on g1 is also running the algorithm and is trying to form a view. As the view manager, it becomes inactive and computes a new viewid and sends invitation messages to g2, g3, g4, and g5. No responses are forthcoming due to the communication failure. It waits in vain for acceptances and eventually times out, remaining inactive.

In this scenario, the algorithm forms a new view, excluding inaccessible cohorts. Although not illustrated, the algorithm also works in the case of including cohorts that become accessible when a failure is repaired.

5.3.2 The case of two active primaries

Suppose a primary is slow to detect a communication failure that has separated it from its backups; it may be slow, for example, because it is doing a lot of work and has not noticed the failure. For example, in Figure 5.8, this might happen to primary P1. So P1 assumes that it can still communicate with its backups and still sees view $v1$. Transactions continue to execute there, unaware that anything is wrong. In the meantime, a more alert backup cohort initiates the view management algorithm and creates a new view $v2$ with a new primary P2. We now have coexisting primaries P1 and P2, each residing in overlapping views $v1$ and $v2$. At first blush, this situation may seem intolerable, since it makes no sense for two primaries to coexist; in fact, we rely on the two-phase commit protocol to prevent incorrect behavior.

There are two cases to consider.

First, the information in P1's buffer may have already propagated to P1's backups before the view change happened. Suppose P1 receives a *prepare* message from the coordinator, after the view change. Since needed information is already at the backups (*force-to* returns immediately), P1 will reply *ok* to the coordinator. When P1 receives the *commit* message from the coordinator, P1 tries to force its buffer to the backups and fails. P1 rejects the commit message. Ultimately, the coordinator's *commit* message will find its way to the new primary P2, where the topaction will finally commit. The prepare succeeded at P1, but the commit could not. A variation on this is the following.

Suppose P1 receives a *prepare* message from the coordinator, forces its buffer, and replies *ok*, all before the view change. As just discussed, the prepare can succeed at P1, but the commit cannot.

Second, suppose all information on behalf of a preparing action is not known to have propagated to the backups before the communication failure. The prepare cannot succeed because P1 will discover that, as part of preparing, it cannot force the event records to its backups in view *v*1 for the preparing topaction. Ultimately, the prepare message will be sent to P2; it will succeed there just in case all its effects survived the view change.

5.3.3 Concurrent view managers

If more than one cohort detects a change in the communication capability within a guardian group, several cohorts may become view managers simultaneously. Clearly, this is inefficient, but any optimization cannot guarantee that only one cohort can act as view manager, as we shall see in a later section. Our view management algorithm handles this case of multiple concurrent view managers in the following way.

Recall that viewids generated by different cohorts are distinct; we achieve this by making a *gid* a part of the viewid. A viewid consists of two parts, a counter *cnt* and the identity of the cohort initiating the view change. Gids are themselves unique, and we can use them to break ties between different cohorts computing a new viewid. Viewids are compared using the $\prec$ relation:

$$v_1 \prec v_2 \equiv (v_1.cnt < v_2.cnt) \lor (v_1.cnt = v_2.cnt \ \land \ v_1.gid < v_2.gid).$$

In the previous example, let us imagine that g1 through g5 are labeled in increasing order. Suppose cohorts g2 and g3 start up as view managers. g2 computes <2, g2> and g3 computes <2, g3>. Both send invitation messages to everybody else in the configuration. Now things suddenly become a little more complicated.

1. g2 receives an invitation from g3. Since <2, g3> $\succ$ <2, g2>, g2 accepts the invitation, sets its *max_viewid* to <2, g3> and stops acting as view manager. It sends accept(g2, <v1, 5>, <2, g3>, false) to g3.

2. g3 receives an invitation from g2. Since <2, g2> $\prec$ <2, g3>, g3 knows of a higher viewid, so it ignores the message from g2.

3. g4 and g5 receive invitation messages from both g2 and g3. What do they do? They might receive the messages in different orders, that is, g4 might receive <2,

g2> and then <2, g3>, and g5 might receive <2, g3>, and then <2, g2>, but that scarcely matters.

- Since the incoming viewid <2, g2> from g2 is bigger than *max_viewid* <1, g1>, g4 stores it in *max_viewid.* It sends accept(g4, <v1, 5>, <2, g2>, false) to g2.
- Since the incoming viewid <2, g3> from g3 is bigger than *max_viewid* <2, g2>, g4 stores it in *max_viewid*, and sends accept(g4, <v1, 5>, <2, g3>, false) to g3.
- Since the incoming viewid <2, g3> from g3 is bigger than *max_viewid* <1, g1>, g5 stores it in *max_viewid*, and sends accept(g5, <v1, 5>, <2, g3>, false) to g3.
- g5 compares its *max_viewid* <2, g3> with the incoming <2, g2>. g5 ignores the message since <2, g2> is smaller than what it currently knows.

g4 accepts g2's invitation to join its view and then accepts g3's invitation; if g4 had received invitations in the opposite order, it would have ignored g2's invitation since g2's viewid is smaller. Thus, g3 prevails. g5 accepts g3's invitation, but ignores g2's since g2's viewid is smaller. Again, g3 prevails. Thus, g3 is the view manager. Clearly, the higher viewid prevails in the absence of additional failures. Anything smaller is rejected.

In other words, no matter what order the messages arrive in the outcome is the same: g3's new viewid is the one that prevails because its viewid is larger. Since a view manager has been selected, this case reduces to the previous, simple one, and the protocol proceeds as above.

5.4 Performance

In this section we discuss the performance of the view management algorithm. We assume that a single failure or recovery event triggers the algorithm; there are no additional failures or recovers and no lost messages. In the simple case, only one cohort acts as view manager. Next, we present several policies to prevent multiple managers from simultaneously starting a view change, again assuming no additional failures or recoveries. Then we comment on the problem of lost messages and present some optimizations.

5.4.1 Simple case

When failures or recoveries are detected by the system, the view management algorithm runs in each affected guardian group. If a new view can be formed, the algorithm requires relatively little message-passing in the simple case of one view manager. We analyze the time and message complexity.

Time complexity. The time complexity of our view management algorithm requires one round of messages, a single message, plus a single write to stable storage.

1. The manager sends *invite* messages.

2. The underlings reply with *accept* messages.

3. The view manager sends a single *newview* message to the cohort selected as the new primary.

In the case in which the view manager is the primary, no *newview* message need be sent; the algorithm requires just one round of messages plus a single write to stable storage.

Message complexity. To analyze the message complexity, let n be the number of underlings, so the total number of cohorts is $n + 1$, including the primary. Thus, n invitation messages are sent to the underlings and at most n acceptance messages are sent to the view manager. Then a single message may be sent to the new primary. Thus, the algorithm uses at most $2n + 1$ messages. In the case in which the new primary is not the same as the view manager, only $2n$ messages are needed.

5.4.2 Preventing concurrent view managers

The algorithm tolerates several cohorts starting up as view managers simultaneously. Having several managers at once will slow things down, since there will be more message traffic, but the slowdown will be slight. We can, however, avoid these parallel view changes to some extent by various policies.

One possible policy is to impose a static order on the cohorts in a configuration. Based on its position in this ordering, each cohort waits longer than its predecessor, thus giving its predecessor a chance to take over as the view manager. For example, suppose we have the following configuration {g1 g2 g3 g4 g5} consisting of an ordered set of gids. The idea is to give the predecessor a chance to take over as the view manager and inform the others. That is, g1 immediately begins executing the algorithm, inviting the others to join its view. In the meantime, g2 waits a certain amount of time that is longer than g1's; if g2 has not received an invitation from g1 after the waiting time has elapsed, then g2 takes over as view manager and sends invitation messages; and so on.

Most of the time this "backoff" strategy will lead to only one cohort acting as view manager (unless the configuration is split due to a network partition). But the strategy does not guarantee that only one cohort will be view manager. Due to timing problems it is possible that two or more cohorts might start up as view managers, a situation that can arise in the following way. Suppose g1 runs on a slow node relative, say, to g2's node. g1 becomes the view manager and sends out *invite* messages to the other cohorts. In the meantime g2's fast node figures it has waited long enough and starts the view management algorithm. Two view managers now coexist; the cohort with the higher viewid prevails, while the other becomes an underling. Problems raised by timing difficulties can manifest themselves in other ways, too. For example, g1's invitation to g2 might be delayed by the network or it might be lost entirely. In either case, g2 would start up as view manager.

Another, better technique is to give priority to the primary of a view. If the primary is lost, the backups will take over in some predetermined order; this order could either be static or set up when the view was created. Inactive cohorts will delay waiting for an active cohort to take over. Again, such a strategy does not guarantee that only one cohort acts as a view manager.

5.4.3 Lost messages

Recall that the **send** statement does not send messages reliably. For example, the *newview* message sent by a view manager to the new primary might be lost. To avoid waiting forever when messages do not arrive, we set a timeout; when the timeout expires, this cohort becomes the view manager, starting a new view change. Starting a view change just to mask a lost message is not efficient, however. Instead, messages should be retransmitted.

5.4.4 Other optimizations

Not all view changes described above really need to be done. One special case occurs when a primary notices that it cannot communicate with a backup, but it still has a sub-majority of other backups. In this case, the primary can unilaterally exclude the inaccessible backup from the view. Similarly, the primary can unilaterally include a backup in its current view.

5.5 Robustness and Making Progress

In our algorithm we assumed that most of a cohort's state was volatile. Such an assumption means that if a majority of cohorts are crashed "simultaneously," we may lose information about the guardian group's state. Here we view a cohort as crashed if either it is really down, or if it has recovered from a crash but its *up_to_date* variable is false. Note that a catastrophe does not cause a group to enter a new view missing some needed information. Rather, it causes the algorithm to never again form a new view.

Whether it is worthwhile to worry about such catastrophes depends on how likely they are and the importance of the information in the group state. The considerations here are similar to decisions about when it is necessary to store information in stable storage in a non-replicated system, except that replication makes the probability of catastrophe smaller to begin with.

If protection against catastrophes is desired, there are various techniques that could be tried. We might have stable storage in use only at the primary. We might supply each cohort with a uninterruptible power supply and have them write information to non-volatile storage in background. Or, we might force events to all backups, thus decreasing the situations in which a new view cannot be formed.

5.6 Discussion

After some event, either a failure or a recovery, triggers the algorithm, the algorithm proceeds unimpeded and eventually terminates with a new view, as long as no additional such events occur. We assume that these events are rare, since otherwise the system would do no useful work but instead would spend all its time responding to such events. Such an assumption is reasonable as long as node and communication failures and recoveries are rare events. Hence, the time between these events is large enough that the algorithm will eventually terminate, forming a new view.

The algorithm does not tolerate decisions being made too quickly. For example, suppose a manager waits only until it hears from a sub-majority even though there are other cohorts that could respond. This would result in excluding those other cohorts from the new view, which in turn will mean another round of view changing will occur shortly. If that next view change also excludes some potential members, that will lead to yet another view change, and so on. To avoid such a situation, a view manager should use a fairly long timeout while it waits to hear from all cohorts that the probe messages indicate should reply.

If the same cohort is the primary both before and after the view change, then no user work is lost in the change. Otherwise, we guarantee the following: Topactions that prepared in the old view will be able to commit, and those that committed will still be committed. Topactions that had not yet prepared before the change may be able to prepare afterwards, depending on whether the completion events of their handler calls are known in the new view. Aborts of topactions may have been forgotten, but delivery of abort messages is not guaranteed in any case; recovery from lost messages was mentioned in Section 4.3.4. To minimize disruption while a view change is happening, queries can be answered by any cohort that knows the answer.

Whenever possible, our algorithm chooses the primary of the last view to be the new primary. This is a good strategy because it makes the algorithm run quickly and ensures that the least amount of work is lost; even handler calls that were running before the view change can continue to run after the change. However, in some systems there may be a favored cohort that should be the primary whenever it is a member of the new view. For example, that cohort may run on a more powerful node than the others. Such a policy matches the needs of some applications. To accommodate such a requirement, we would need to change our algorithm since the new primary may need to read the current state from one of the backups in phase two. The modified algorithm is probably best combined with a strategy that makes the favored primary most likely to be the view manager; in this case, one round plus two messages are needed before the favored primary could become active after it had crashed or become inaccessible because of a partition.

The policy would not necessarily cause loss of information: if the old primary is a member of the new view, all its events will survive into the new view. However, work in progress at the old primary, including aborting active transactions for which it is the coordinator, would be lost in the change, unless some additional mechanism is included.

6 Related Work

In this chapter we discuss the relationship of our approach to other work on replication and on view changes.

6.1 Voting

The best known replication technique is voting [Gifford 79]. Gifford presents a simple and elegant protocol for maintaining the consistency of replicated data in a distributed computer system.[1] The basic idea of the protocol rests on the notion of *quorum intersections.* Each copy of a replicated data item is assigned some number of votes. To read a data item, a transaction must collect a read quorum of votes; to write a data item, it must collect a write quorum of votes. To maintain the consistency of the replicated data, these read and write quorums must satisfy two constraints. First, read and write quorums must intersect, guaranteeing that any read quorum has a current copy of a data item. Second, write quorums must intersect, imposing an order on updates. Together, these two rules ensure one-copy serializability. The protocol has several additional benefits: it continues to operate correctly even if some copies are inaccessible, it is possible to change a data item's performance and reliability characteristics by altering quorum

[1]The replication scheme is built on top of a transaction system, which is a major reason for its simplicity.

sizes, and it also copes with partitions without explicit detection. Herlihy [Herlihy 86] extended Gifford's voting protocol to take advantage of operation semantics, thus making the protocol more efficient.

Our method is faster for write operations because we communicate with only the primary; with voting we must write to at least a simple majority of copies. Also, we avoid deadlocks that can arise if messages for concurrent updates arrive at the replicas in different orders. Our method will also be faster for read operations if the reads take place at several replicas. If read operations take place at only one replica, voting may outperform our method because reading can occur at any replica, while reading in our scheme must happen at the primary, which could become a performance bottleneck. On the other hand, the real source of a bottleneck is a node, not a cohort, and we can organize our system so that primaries of different groups usually run on different nodes. Furthermore, the system can be configured to place primaries at more powerful nodes most of the time. This organization could lead to better performance than voting.

6.2 Virtual Partitions

Our view change protocol is a simplification and modification of the original virtual partitions protocol [El Abbadi 85], a variation on Gifford's weighted voting. Like our notion of views, virtual partitions attempt to track real changes in the network topology as closely as possible without being constrained by the need to cope with changes instantaneously. A virtual partition is a set of processors that have agreed that they can communicate with each other and further agree that they will not communicate with any processors outside the partition. Although communication with processors outside a virtual partition may be physically possible, this communication may not be initiated until a special protocol is run to form a new virtual partition. A logical data item is *accessible* in a virtual partition that includes a majority of its sites. A transaction reads or writes only those logical data items that are accessible in its virtual partition. The principal advantage of this scheme is that transactions can always read from a single copy; this advantage comes at the expense of an update sub-protocol that updates every item in the database when partitions are repaired.

The protocol requires three phases. In phase 1, a processor starts a view change by sending a "newvp" message to every processor in the network; it waits for "ok" messages to flow back. In phase 2, that processor sends "commit" messages to all accepting processors. In phase 3, each processor in the newly formed virtual partition updates its local copies of replicated data objects with the most recent values; it does so by sending "read" messages to all copies in parallel and waiting for responses. Our algorithm is more efficient because it requires only one and one-half phases. We avoid extra work by using viewstamps in phase 1 (the first round) to determine what each cohort knows.

Virtual partitions force transactions that were active across a view change to abort. For example, a transaction that did a remote procedure call in the old view will not be able to prepare in the new view. We use viewstamps to avoid the abort and we rely on the fact that knowledge of later events implies knowledge of earlier ones.

6.3 Isis

A different approach to replication is taken in Isis [Birman 85]. Because Isis's view change protocol does not tolerate partitions, it only works in a local area network; in this sense, it is is not comparable with viewstamped replication. However, it does have some interesting characteristics. As with our technique, Isis has resilient modules consisting of instances [Birman 85]. Like Argus guardians with handlers, these modules have state that is modified only by calling their operations, issued as RPCs. A resilient module guarantees that computations in progress complete as long as at least one component is operational. Computations run as atomic actions and satisfy the one-copy serializability correctness criterion.

Isis uses an unusual sort of replication scheme to implement resilient modules. Rather than designating one component as the primary to which all clients direct all requests, any component in a resilient module can act as the primary on a *per request* basis. This primary is called the *coordinator* for the request and the other components are called *cohorts*. Each component of the resilient module knows the other operational components of the module. Since all components of a resilient module can be coordinators for different client requests, these invocations must be synchronized.

Isis uses two-phase locking for concurrency control. If the operation is a read, a component acquires a read lock locally and performs the operation. If the operation is a write, a component first acquires locks at all operational components before doing the update. The locks are acquired using an expensive two-phase algorithm that prevents deadlocks in the case of concurrent writes. After acquiring the needed locks, the coordinator performs the operation. The correct serialization order with respect to a failure or recovery event is guaranteed by imposing the same relative order at all the components, and by preserving all read and write locks across failures. Three atomic broadcast protocols were proposed [Birman 87] to enforce varying ordering constraints.

Effects of reads and writes are communicated to other components in background mode, are piggybacked on reply messages, and accompany further client messages such as prepare and commit messages. This means that information needed to process these later messages is always available to the component that receives them. The disadvantage of their scheme, however, is the large amount of extra information flowing on every message, the high storage overhead, and the difficulty in garbage-collecting that information. Isis works only in a local area net, both because their garbage collection algorithm depends on broadcast and because the protocol cannot tolerate partitions.

Our method avoids these problems at the cost of possible delay at prepare time (to force the buffer), and at the cost of an occasional abort when there is a view change. The viewstamps in our method represent the information flowing in Isis. Since the viewstamps only indicate that certain events have occurred, but not what these events are, we must sometimes wait for information about events to arrive in buffer messages. We must sometimes abort a transaction because information about events is lost in a view change.

6.4 Circus

Cooper [Cooper 84, Cooper 85] proposed *replicated remote procedure call* as the mechanism with which to construct highly available distributed programs. Each program module is replicated; the set of replicas is called a *troupe*. The troupe behaves as a single logical module with state that may change over time.

In a distributed program made up of troupes, a remote procedure call from a client to a server is actually a replicated remote procedure call from client troupe to server troupe. Each client troupe member makes a one-to-many call to all the server troupe members. Each server troupe member, which will receive as many calls as there are client troupe members, executes the call (and may make calls to other troupes) exactly once, possibly changes its state, and returns a result. Changing the individual states changes the collective state of the server troupe.

To guarantee the single-view image, troupe members must be consistent and must behave in a deterministic fashion: two replicas in the same state must execute the same remote call in the same order, produce the same side effects, and return the same result. Requiring programs to be deterministic is severely restrictive because it reduces concurrency in each module and is unrealistic because it burdens the programmer with the responsibility for finding all sources of non-determinism and overcoming them. To weaken the assumption of complete determinism the application programmer can define a *collator* procedure that reduces a set of messages to a single message; this violates replication transparency because the programmer is now aware that a module is replicated.

To handle the problem of concurrent, replicated remote calls to the same server from different clients, Cooper introduced transactions. Independent serialization of transactions at each troupe member is insufficient; to preserve troupe consistency, concurrent calls from different clients must not only be serialized by each server troupe member but they must be serialized in the *same order* at all server troupe members. Cooper proposed two protocols to solve this problem. The *troupe commit protocol* detects any attempt by troupe members to serialize transactions differently and transforms such attempts into deadlocks. It operates on the assumption that concurrent transactions are unlikely to conflict; this protocol suffered from starvation under heavy loads. The *starvation-free* protocol does not introduce any additional chance of deadlock. It uses an ordered broadcast protocol that guarantees that concurrent broadcasts are never interleaved, and requires a deterministic local concurrency control protocol at each troupe member. It has the disadvantage that it limits the potential concurrency.

The replicated remote procedure call mechanism is expensive during normal system operation, exhibiting high overhead. Each replicated call from an m-member client troupe to an n-member server troupe requires $m \cdot n$ messages in both directions; the execution time of each call is determined by the slowest member in each troupe. The mechanism wastes computational power because all replicas are involved in executing each remote call; the time per call increases linearly with the size of the troupe. Its chief virtue is that performance in the presence of failures and recoveries is essentially unaffected.

A further problem is that in the presence of partitions, the state of troupe members in different partitions will become inconsistent. To solve this problem, Cooper suggests that each troupe member receive a majority of the expected set of messages before computation is allowed to proceed there. After the partition is repaired it is not clear how the divergent states are reconciled.

Our method is simpler than Cooper's, requires far fewer messages, and imposes no determinism requirement on programs.

6.5 Tandem's NonStop System

Tandem's NonStop System [Bartlett 78, Bartlett 81] is the first general-purpose, commercially available, fault-tolerant computer system that was designed for on-line transaction processing and that could expand over its lifetime to accommodate growth of applications. Tandem modified conventional hardware so that all components are backed up in hardware; for example, there are the dual interprocessor bus, dual port disk controllers, and mirrored disks (stable storage); if any single piece of hardware fails, the corresponding backup can take over its function. A Tandem node consists of two to sixteen processors, each with its own memory, which communicate via a dual interprocessor bus; a Tandem network would consist of many such nodes.

At the software level, process-pairs and messages are the abstractions that hide the boundaries of the processors. *Process-pairs* are used as the uniform mechanism for accessing system resources, such as I/O devices, in a fault-tolerant fashion. The process-pair consists of two processes, each of which runs in a distinct processor within the same node. The *primary* process is active and sends information via checkpoint messages to it *backup* process, which is ready to take over control whenever the primary process fails. For example, each disk volume (mirrored drives) is accessed through a process-pair running in the two processors physically connected to the controllers. This process-pair is called the *disk process*.

Bartlett suggested that process-pairs could be used to make application programs fault-tolerant. Before transactions were introduced, fault-tolerant application programs were coded in this fashion to preserve database consistency. But writing these programs was hard for two reasons. First, the programmer had to insert, by careful design, the appropriate checkpoint statements in his programs. Second, process-pairs always carried a computation to completion. To handle failures, the programmer also had to write code to backout a computation. Organizing application programs as transactions to handle failures automatically was a better idea; to this end Tandem introduced the Transaction Monitoring Facility (TMF) [Borr 81].

The application programmer typically brackets a sequence of operations with BEGINTRANSACTION and ENDTRANSACTION, indicating that the sequence should be treated as one transaction. Transactions update a database by sending requests to Disk Processes (DP) that maintain lock information for those database records and files residing on its volume only. Each DP is a process-pair that synchronizes concurrent access to the database. During transaction processing, a request is sent to the primary DP, which locks the record. It alters the record in a cache buffer and writes the *before* and *after* images to its internal log. A distinguished DP that maintains the log on stable storage is called the Audit Disc Process (ADP).

During phase one of two-phase commit, the primary of each DP that participated in the transaction ensures that its log records have been flushed first to its backup and then to the ADP. In phase two, the coordinator writes a commit record to the ADP; the ADP is flushed to disk. Any failures before the commit record makes it to disk causes transaction UNDO; a failure after this point causes transaction REDO. The transaction is committed when the commit record is on disk; all DPs release locks.

Process-pairs and TMF together make it possible for application programs to continue execution even if there are hardware faults [Helland 85, Gray 86]: all uncommitted transactions associated with a failed primary process are aborted and then restarted with the backup process as the new primary. This is a new design and is described by Borr [Borr 84]. The new primary "falls back" and aborts some transactions. This new implementation required half as many messages and a fifth as many bytes [Helland 85] as in the original design of the DP, in which one always rolled forward after a crash and continued execution.

Tandem's NonStop system survives only a single failure and requires that a process-pair reside at a single node. If a processor crash causes a backup to takeover as the new primary, that primary runs without a backup until the crashed processor comes back on-line. If these contraints are acceptable, then this method is efficient. Our replication method is more general.

6.6 Auragen

Auragen [Borg 83] is fault-tolerant computing system that is used in an on-line transaction processing environment. It is based on the notions of primary/backup process pairs, the three-way message send, automatic synchronization of the primary and backup, and user determinism. The design goals of this system are similar to Tandem's but were realized in a different manner.

Like Tandem, Auragen's hardware base contains redundant components. The *cluster* is the basic processing unit. The Auragen 4000 computer consists of two to thirty-two clusters connected by a dual high-speed intercluster system bus. Each cluster contains between three and seven Motorola M68000s and a large shared memory. Two processors in the cluster run user and system server processes to handle input/output via messages and global system resources. Other processors control intercluster message traffic, communication ports, and dual-ported peripheral devices. Overlapping the execution using different processors is claimed to lead to more efficient overall operation, even though the shared memory might be a bottleneck.

Processes can run backed-up or not. For the purposes of this description, we assume that each process consists of a primary and a backup, which execute in different clusters and communicate by passing messages. Whenever the primary crashes, its backup is notified and takes over execution as the new primary.

The system automatically brings the backup process up to date with its primary periodically, a procedure called synchronization. Upon failure of the primary, the backup rolls foward, recomputing based on the messages in its queue (received from the primary) since it was last synchronized with the primary. User processes are required to be deterministic because the backup process must reconstruct a state that is the same as the primary's state before the primary failed. The rule for determinism states that if two processes start out in identical states and receive the same set of messages in precisely the same order, then after reading those messages and computing based on them, their final states will be identical. To enforce this rule, a message in the Auragen system is sent by a sender process to three destinations atomically: the receiver process, its own backup process, and the receiver's backup process. (The hardware and software guarantee that message delivery is atomic.)

The Auragen system survives only a single failure and, like Tandem, requires that a process-pair reside at a single location. In addition, Auragen scales poorly because the message-passing mechanism depends on special hardware support for interprocess communication. New backup processes are not automatically created when old backups fail or a backup takes over as the new primary. Our replication method is more general.

7 Conclusions

This dissertation has presented a new replication method to solve the problem of constructing highly available computer-based services. We believe that programmers should write distributed programs without worrying about availability; the underlying language implementation uses our replication technique to replicate modules automatically. The resulting services implemented by these distributed programs are highly available. Our method performs well in the normal case, does view changes efficiently, and loses little information in a view change.

In the remainder of this chapter, we summarize our accomplishments and suggest directions for future work.

7.1 Summary

Our replication algorithm works out for the first time the details of a primary copy replication scheme, which others have only hinted at. We take advantage of the method's intuitive appeal: placing the primary copy where it is needed or where there is a more powerful node, avoiding synchronization problems, and incurring low delay when executing transactions. As discussed in Chapter 4, the performance of our method is comparable to that of a system in which modules are not replicated, and is better than other replication methods.

Each replicated module consists of several instances, called cohorts, constituting a module group. One cohort is designated the primary; the others are backups. The primary is responsible for the module group's activity; it executes remote procedure calls and modifies its state. When remote procedure calls complete, the primary sends the effects of the calls to its backups in background mode.

Since the primary only communicates with the backups in background mode, the effects of some calls may be lost after a view change. If the effects of all calls made a transaction are known at the new primary, then no information is lost and the transaction can commit; otherwise, it must abort. Furthermore, if transactions commit, we guarantee that their effects are not lost in subsequent view changes. We use viewstamps, a special kind of timestamp, to represent how much a cohort "knows" about the effects of transactions that have run. The viewstamp history represents the sequence of view changes seen by a cohort. Each member of the sequence is a viewstamp; for each viewstamp *vs* in the history, the cohort's state reflects each event in the view of *vs.id* whose timestamp is less than or equal to *vs.ts*. What a cohort *does* know and what it *should* know are used to determine whether transactions can commit or must abort.

Our view management algorithm reorganizes the cohorts of a configuration to form new views under certain conditions. It is efficient, since it requires just one round of messages (invitation and acceptances) and one message (to notify the new primary). Viewstamps are again used here; they indicate which cohort knows the most. The cohort with the largest viewstamp is chosen as the new primary. The new primary's state is used to initialize the state of all other cohorts in the new view.

Our view management algorithm is highly likely not to lose work in a view change. Our policy of choosing the primary of the last view to be the new primary whenever possible avoids losing work altogether; even remote calls that were running before the view change can continue to run afterwards. Note that the probability of aborts can be decreased further if desired. There is a tradeoff here between loss of information in view changes and speed of processing calls. For example, if "completed-call" records were forced to the backups before the call returned, there would be no aborts due to view changes, but calls would be processed more slowly.

The correctness of our algorithm depends on the interaction of transaction processing and the view management algorithm. Transactions must still be serializable and recoverable. Transaction processing guarantees that a transaction can commit only if all its events are known to at least a majority of cohorts. The view management algorithm guarantees that events known to a majority of cohorts survive into subsequent views. Thus, events of committed transactions will survive view changes.

7.2 Directions for Future Work

In this section, we suggest some areas for further work.

Implementation. To understand how well the replication algorithm performs, it must be implemented, and performance measurements taken. We are planning to implement the algorithm as part of the Argus system and to run experiments to measure its performance.

Optimizations. Another area of interest is optimizations. We point out some general directions that further research might take.

1. Efficiently updating the backup state. We send the entire guardian state from the new primary to the backups. Since this is clearly inefficient, we must investigate other methods of updating the backup's state. For example, if the primary knew the old viewstamps for the backups, and if it had recorded information about events and their viewstamps, then it could send the difference between what it knows and what the backups know to the backups.

2. Event records that arrive at backups can be performed immediately and the state updated, or they could be stored and then performed at a convenient time, say, when the committed or aborted record arrives. It is a matter of future research to understand the right tradeoff between processing during normal operation and after a view change.

3. Avoiding unnecessary view changes. View changes really need to happen only when the primary becomes inaccessible, or the current view loses enough members that it no longer constitutes a majority. A view change need not be done if a backup fails and the view still has a majority, for that backup can be excluded unilaterally. Similarly, a view change is unnecessary if a backup becomes accessible. These protocols need to be worked out.

4. Garbage-collecting the viewstamp history. Over time, the viewstamp history can grow without bound. To remove a viewstamp from the history, the system might, for example, wait until all transactions that depended on the view of that viewstamp have committed. How this is determined is a matter of future research.

Performance/cost model. Comparing the performance of our replication method with that of a conventional, non-replicated system is straightforward. Comparing our scheme with other replication methods is much harder because other methods make different assumptions and have different goals. It would be interesting to develop a performance/cost model that provided a basis for comparison.

Reconfiguration. After defining a configuration initially, we may wish to change it. For example, we might add backups to increase the resiliency of the group to failure or we might delete old backups deemed permanently inaccessible. This process of changing the configuration is called *reconfiguration.* We need to invent extensions to our method to support reconfiguration.

Dealing with catastrophes. In designing our algorithm, we chose to make as little use of stable storage as possible because we were interested in understanding the extent to which having several replicas eliminated the need for stable storage. We found that catastrophes (loss of a group's state) could sometimes occur in our system that would not happen if more information had been recorded on stable storage. Whether we should worry about catastrophes depends on how likely they are to happen, how important the group's state is, and the environment in which the system runs. Engineering decisions must be made here. The probability of a catastrophe depends on the configuration, such as whether the cohort's nodes are failure-independent. To reduce this probability, the algorithm can be modified in various ways. What these ways are is a matter of future research.

Formal proof of correctness. We have stated some conditions for correct operation of our system. It would be interesting to characterize precisely what our algorithm achieves under certain failure assumptions and what invariants must be preserved by our implementation. In other words, we should undertake a formal proof of correctness of the replication algorithm. These conditions are safety properties that ensure that nothing bad ever happens during execution of the algorithm. Of equal importance is stating liveness properties that eventually something good will happen; in particular, what guarantees can we make that another view will eventually be formed?

Viewstamps. Viewstamps are an interesting subject in their own right. First, we might investigate how viewstamps could be used in a non-replicated system. For example, in such a system records containing the effects of calls could be written to stable storage in background mode; these records, like our event records, would contain viewstamps. When the prepare message arrives, it would only be necessary to force the records; no delay would be encountered if the records had already been written. A crash would not cause active transactions to abort automatically; instead, queries would be sent to coordinators to determine the outcomes. The result would be a system that is more tolerant of crashes (by avoiding aborts) and also faster at prepare time.

Second, we can investigate how viewstamps might be used in conjunction with other replication methods. For example, our technique can be used with voting when writes are done at all members of a view. Just as we use viewstamps, in such a system timestamps that are assigned when transactions commit could be used to determine which replica has the most information about transaction commits (the timestamps would not contain information about the state of active transactions). Systems in which writes only go to a majority are more difficult to optimize in this way since there is usually no cohort whose state contains at least as much information as the state of any other cohort. A total order on viewstamps would be costly to implement with voting since there is no single place (like our primary) to generate the viewstamp. Whether we could use multipart viewstamps [LiskovLadin 86, Ladin 88] is a matter of further investigation.

References

[Alsberg 76] Peter A. Alsberg and John D. Day. "A Principle for Resilient Sharing of Distributed Resources." In *Proceedings of the 2nd International Conference on Software Engineering*, pages 627–644, October 1976. Also available in unpublished form as CAC Document number 202 Center for Advanced Computation University of Illinois, Urbana-Champaign, Illinois 61801 by Alsberg, Benford, Day, and Grapa.

[Bartlett 78] Joel F. Bartlett. "A 'NonStop' Operating System." In *Eleventh Hawaii International Conference on System Sciences*, pages 103–117, January 1978.

[Bartlett 81] Joel F. Bartlett. "A NonStop Kernel." In *Proceedings of the 8th ACM Symposium on Operating System Principles*, pages 22–29, December 14-16 1981. Appeared in a special issue of SIGOPS Operating System Review, Vol. 15, No. 5. Held at Asilomar Conference Grounds, Pacific Grove, California.

[Bernstein 83] Philip A. Bernstein and Nathan Goodman. "The Failure and Recovery Problem for Replicated Databases." In *Second ACM Symposium on the Principles of Distributed Computing*, pages 114–122, August 1983.

[Birman 85] Kenneth P. Birman, Thomas A. Joseph, Thomas Rauchle, and Amr El Abbadi. "Implmenting Fault-tolerant Distributed Objects." *IEEE Transactions on Software Engineering*, 11(6):502–508, June 1985.

[Birman 87] Kenneth P. Birman and Thomas A. Joseph. "Reliable Communication in the Presence of Failures." *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[Borg 83] Anita Borg, Jim Baumbach, and Sam Glazer. "A Message System Supporting Fault Tolerance." In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 90–99, October 10-13 1983. Appeared in a special issue of SIGOPS Operating System Review, Vol. 17, No. 5. Held at the Mt. Washington Hotel, Bretton Woods, New Hampshire.

[Borr 81] Andrea J. Borr. "Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing." In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 155–165, September 9-11 1981. Held in Cannes, France. Sponsored by ACM SIGMOD SIGBDP and SIGIR IEEE and INRIA.

[Borr 84] Andrea J. Borr. "Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach." In *Proceedings of the Tenth International Conference on Very Large Data Bases*, pages 445–453, August 1984. Held in Singapore, Malaysia.

[Cooper 84] Eric C. Cooper. "Replicated Procedure Call." In *Proceedings of the 3rd Annual ACM Symposium on the Principles of Distributed Computing*, pages 220–232, August 27-29 1984. Sponsored by SIGACT and SIGOPS. Held in Vancouver, British Columbia, Canada.

[Cooper 85] Eric C. Cooper. *Replicated Distributed Programs.* Technical Report UCB/CSD 85/231, U. C. Berkeley, EECS Dept., Computer Science division, May 1985. Ph.D. thesis.

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

[El Abbadi 85] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. "An Efficient, Fault-Tolerant Protocol for Replicated Data Management." In *Proceedings of the 4th ACM SIGACT/SIGMOD Conference on Principles of Data Base Systems*, 1985.

[El Abbadi 86] Amr El Abbadi and Sam Toueg. "Maintaining Availability in Partitioned Replicated Databases." In *Proceedings of the 5th ACM SIGACT/SIGMOD Conference on Principles of Data Base Systems*, 1986. Held at the Hyatt Regency Hotel, Cambridge, Massachusetts, March 24-26, 1986.

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

[Fowler 85] Robert J. Fowler. *Decentralized Object Finding Using Forwarding Addresses.* Technical Report 85-12-1, University of Washington, Department of Computer Science, Seattle, Washington, December 1985.

[Gifford 79] David K. Gifford. "Weighted Voting for Replicated Data." In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, December 10-12 1979. Appeared in a special issue of Operating Systems Review, Vol. 13, No. 5. Held at Asilomar Conference Grounds, Pacific Grove, California.

[Gifford 84] David K. Gifford and Alfred Z. Spector. "Case Study: The TWA Reservation System." *Communications of the ACM*, 27(7):650–665, July 1984.

[Gifford 85] David K. Gifford and James E. Donahue. "Coordinating Independent Atomic Actions." In *Proceedings of IEEE CompCon85*, pages 92–95, Feburary 1985.

[Gray 76] James N. Gray, Raymond A. Lorie, G.F. Putzolu, and Irving L. Traiger. "Granularity of locks and degrees of consistency in a shared data base." In *Modeling in Data Base Management Systems*, pages 365–394, Elsevier North-Holland, New York, 1976.

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

[Gray 86] Jim Gray. "Why Do Computers Stop and What Can Be Done About It?." In *Proceedings of the Fifth IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, January 13–15 1986. Held at the Marriott Hotel, Los Angeles, CA.

[Helland 85] Pat Helland. "Transactions and Fault Tolerance." February 14, 1985. Helland is with Tandem Computers, Inc., Cupertino, California. Unpublished paper.

[Henderson 82] Cecilia Henderson. "Locating Migratory Objects in an Internet." 1982. Master's thesis, MIT Laboratory for Computer Science. Available as Computation Structures Group Memo 224, MIT LCS.

[Herlihy 86] Maurice P. Herlihy. "A Quorum-Consensus Replication Method for Abstract Data Types." *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[Hwang 87] Deborah J. Hwang. *Constructing a Highly-Available Location Service for a Distributed Environment.* Master's thesis, M.I.T. Laboratory for Computer Science, November 1987. Master's thesis.

[Ladin 88] Rivka Ladin, Barbara Liskov, and Liuba Shrira. *"A Technique for Constructing Highly-Available Services"*. Technical Report MIT/LCS/TR-409, M.I.T. Laboratory for Computer Science, Cambridge, MA, January 1988. To be published in Algorithmica.

[Lamport 82] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem." *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[Lampson 81] Butler W. Lampson. Atomic Transactions. In *Distributed Systems: Architecture and Implementation*, chapter 11, pages 246–265, Springer-Verlag Berlin, 1981.

[Liskov 87] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Schielfer. "Implementation of Argus." In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 111–122, November 8–11 1987. Appeared in a special issue of SIGOPS Operating Systems Review, Vol. 21, No. 5. Held at Austin, Texas. Also available as Programming Methodology Group Memo 57, MIT LCS, August 1987.

[Liskov 88] Barbara Liskov. "Distributed Programming in Argus." *Communications of the CACM*, 31(3):300–313, March 1988.

[LiskovLadin 86] Barbara Liskov and Rivka Ladin. "Highly-Available Distributed Services and Fault-Tolerant Distributed Garbage Colletion." In *Proceedings of the Fifth ACM Symposium on the Principles of Distributed Computing*, pages 29–39, August 11-13 1986. Held at the Skyline Hotel, Calgary, Alberta, Canada. Also Programming Methodology Group Memo 48, M.I.T. Laboratory for Computer Science, Cambridge, MA, 1986.

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

[Mullender 85] Sape Mullender and Paul Vitanyi. "Distributed Match-Making for Processes in Computer Networks—Preliminary Version." In *Proceedings of the Fourth ACM Symposium on the Principles of Distributed Computing*, August 1985. Held at Minaki, Ontario, Canada. Sponsored by ACM.

[Nelson 81] Bruce Jay Nelson. *Remote Procedure Call.* Technical Report CMU-CS-81-119, Carnegie-Mellon University, May 1981. Ph.D. thesis.

[Papadimitriou 79] Christos H. Papadimitriou. "Serializability of Concurrent Database Updates." *Journal of the ACM*, 24(4):631–653, October 1979.

[Schneider 83] Fred B. Schneider. "Fail-Stop Processors." In *Digest of Papers from Spring CompCon '83 26th IEEE Computer Society International Conference*, pages 66–70, March 1983. Held in San Francisco California.

[von Neumann 56] John von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In Claude E. Shannon and John McCarthy, editors, *Automata Studies*, pages 43–98, Princeton University Press, 1956.

[Weihl 85] William Weihl and Barbara Liskov. "Implementation of Resilient, Atomic Data Types." *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.

Biography

Brian Masao Oki was born in Inglewood, California, a suburb of Los Angeles, and raised in Gardena, where he attended the Los Angeles public schools. He graduated from Peary Junior High School in June 1973 and from Gardena High School in June 1976. At Gardena High he shared the Science Departmental Award with a fellow student, won the California Savings and Loan League Association's Outstanding Student Award, and received Bank of America's specific field of mathematics award.

Tiring of the University of Southern California after a one-year stint as a freshman, he transferred to the University of California at Irvine, where he majored in Information and Computer Science. He graduated *summa cum laude* with a B.S. in June 1980 and was elected to Phi Beta Kappa.

To avoid working in the real world, he enrolled in graduate schoool in the fall of 1980 at the Massachusetts Institute of Technology. He received the S.M. degree in May 1983, the E.E. degree in June 1985, and the Doctor of Philosophy degree in May 1988, all in Computer Science from the Department of Electrical Engineering and Computer Science. He interrupted his career as a perpetual student by spending the summer of 1983 at what was then the IBM San Jose Reseach Laboratory. He is presently with the Computer Science Laboratory of Xerox's Palo Alto Research Center in Palo Alto, California.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution is unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
MIT/LCS/TR-423

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 Blvd.
Arlington, VA 22217

10. SOURCE OF FUNDING NUMBERS

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

11. TITLE *(Include Security Classification)*
Viewstamped Replication for Highly Available Distributed Systems

12. PERSONAL AUTHOR(S)
Oki, Brian Masao

13a. TYPE OF REPORT
Technical

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Year, Month, Day)*
1988 August

15. PAGE COUNT
99

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Primary copy, replication, viewstamp, view management, high availability, fault-tolerance, transactions, nested transactions, atomicity, distributed computer systems

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

This dissertation presents viewstamped replication, a new algorithm for the implementation of highly available computer services that continue to be usable in spite of node crashes and network partitions. Our goal is to design an efficient mechanism that makes it easy for programmers to implement these services without complicating the programming model. Our replication method is based on a primary copy technique, where one replica is the primary and others are backups, and is integrated into the fabric of an atomic transaction mechanism. Transactions are run only at the primary and need not involve the backups; the primary propagates the effects of transaction processing to the backups in the background. The method exhibits low delay during normal operation, has low overhead, and increases the likelihood that transactions will commit in spite of failures.

When failures occur, replicas are reorganized automatically and a new primary is selected if the old one becomes inaccessible. This reoganization is called a view change and is accomplished by a view management algorithm. Since the primary only (continued..)

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

Unclassified

19. communicates with the backups in background mode, the effects of some processing may be lost after a view change; the affected transactions must abort. If the effects are known at the new primary, then no information is lost and the transaction can commit. Furthermore, if transactions commit, we guarantee that their effects are not lost. A special kind of timestamp, called a viewstamp, allows the algorithm to distinguish these cases inexpensively.