Automatic Replication for Highly Available Services

by

Sanjay Ghemawat

January 1990

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

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

Automatic Replication for Highly Available Services

by

Sanjay Ghemawat

Abstract

Replicating various components of a system is a common technique for providing highly available services in the presence of failures. A replication scheme is a mechanism for organizing these replicas so that as a group they provide a service that has the same semantics as the original unreplicated service. Viewstamped replication is a new replication scheme for providing high availability.

This thesis describes an implementation of viewstamped replication in the context of the Argus programming language and run-time system. The programmer writes an Argus program to provide a service without worrying about availability. The run-time system automatically replicates the service using the viewstamped replication scheme, and therefore makes the service highly available. Performance measurements indicate that this method allows a program to be made highly available without degradation of performance.

Keywords: Replication, Availability, Transactions, Nested transactions, Viewstamped replication, Primary copy replication, Commit protocols

This report is a minor revision of a Master's thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on January 24, 1990, in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science. The thesis was supervised by Professor Barbara H. Liskov.

Acknowledgments

First, I thank Barbara Liskov, my thesis supervisor, for suggesting this topic. Her insights and comments have been a great help throughout my career at M.I.T., and her careful proof-reading significantly improved the presentation of this thesis.

I also thank Wilson Hsieh for applying his great skill at proof-reading to this thesis. He corrected many stylistic problems with the presentation. Bob Gruber and Paul Johnson also read portions of earlier drafts and pointed out several problems. Paul Wang almost read some of the earlier chapters.

The members of the programming methodology group made the work environment interesting and fun. In particular, I thank Brian Oki for many enlightening discussions about the viewstamped replication scheme. My officemates, Debbie Hwang and Carl "The Economist" Waldspurger put up with my messy desk and initiated many interesting discussions. Wilson Hsieh provided science fiction books and numerous other distractions. Bob Gruber occasionally showed up for work. Paul Wang and Anthony Joseph "encouraged" me to go to my office and do some work. Jeff "Just change the source" Cohen cheated at computer games.

I thank Kim Klaudi-Morrison, Joe Morrison, Steve Kommrusch and Alice Sunderland for great Saturdays and countless movie rentals. They provided a much needed outlet from work. Joe and Kim arranged great weekend trips; Alice drove me there, detours and all. Steve demonstrated the pitfalls of showing up for karate class without getting enough sleep. Alok Chopra cooked great meals, drove me around town and played tennis. Steve Gilbert and Mike Villalba were great apartment-mates.

Finally, my heartfelt thanks go to my family, especially my parents, for their love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High availability is essential to many computer-based services. Consider an airline reservation system that handles flight bookings for customers from all over the country. Suppose this system resides on a single computer. A failure of this computer could cripple the airline because customers could not make any reservations or get flight information for the duration of the failure. This loss of service is highly undesirable; we would like to make the reservation service remain available in spite of failures.

A common technique for achieving availability is to replicate the service. For example, if there are copies of the reservation system on several computers, the failure of a single computer could be tolerated because the other copies would remain available. This replication, however, does not come for free. The replicas have to be organized so that as a group they provide a service equivalent to the original unreplicated service. Various replication schemes that achieve this organization are described in the literature: *weighted voting* [Gif79], *quorum consensus* [Her86] and *primary copy replication* [AD76].

This thesis describe an implementation of *viewstamped replication* [Oki88], which is an extension of primary copy replication. This implementation demonstrates that viewstamped replication is an efficient technique for building highly available services. The replication scheme has little impact on the performance of the service, but allows the service to remain available in the presence of failures. Performance measurements that support this claim are given in a later chapter.

This implementation is done in context of the Argus programming language and run-time system [Lis88]. The Argus system has been designed as a tool for easy construction of long-lived services. The Argus run-time system is modified to automatically make services written using the Argus programming language highly available.

1.1 The Replication Scheme

In primary copy replication, the unit of replication is a module. A module is an encapsulation of some resources together with the operations used to access these resources. Each module is replaced by several copies. One of these copies is designated the primary; the others are backups. All module operations are executed at the primary. The effects of these operations are propagated to the backups in the background. If the primary ever fails, a backup takes over and becomes the new primary.

Viewstamped replication extends the primary copy scheme to limit the impact of system failures. Whenever a failure occurs, normal activity is suspended, the replicas are reorganized, and a new primary is selected if the old one is now inaccessible. When this reorganization is complete, normal activities are allowed to resume.

[Oki88] gives a comparison of the viewstamped replication scheme with several well-known replication schemes and systems from the literature (Voting [Gif79], Virtual Partitions [ESC85], Isis [Bir85], Circus [Coo85], Tandem's NonStop System [Bar78, Bar81], and Auragen [BBG83]).

1.2 System Model

The replication method operates in a distributed computer system that consists of many nodes linked by a communication network. The communication network may have an arbitrary topology — for example, many local area networks connected by a long haul network. Processes residing on different nodes can communicate by sending messages over the network. We assume that in the absence of failures, any node can send a message to any other node in the system.

Both the nodes and the communication network may fail. Nodes may crash, but we assume that they are *failstop* [Sch83]; i.e., once a node fails in any manner, it stops all activity. Each node has volatile storage that is lost in a crash. Nodes may also have disks that provide non-volatile storage. Data stored on disks survives most node crashes, but some node crashes may result in disk media failures that destroy this data. Each node also has a small amount of stable storage. Stable storage is a memory device that preserves information written to it with a very high probability [Lam81].

The communication network may drop messages, duplicate them, deliver them out of order, or delay them. In addition, communication link failures might cause the network to partition into isolated subnetworks that cannot communicate with each other. We assume that both node and link failures are repaired.

1.3 Overview

Chapter 2 presents the Argus model of computation which is based on transactions [Gra78], and describes the portions relevant to this implementation.

Chapter 3 gives a general overview of the replication scheme. In particular, the terminology used is presented, and then the replication scheme is described.

Chapter 4 covers my implementation of transactions. It describes the extra processing needed at the clients and the servers to integrate transactions into the replication scheme. Performance measurements of the replicated system are given, and compared with the corresponding measurements for an unreplicated Argus system. A brief comparison of the replicated system with the replicated transaction facility in Isis [Bir85] is also given.

Chapter 5 presents my implementation of the algorithm used to handle system failures. This algorithm involves communication between the replicas to select a new primary. The performance of this algorithm is discussed, and some future optimizations are presented.

Chapter 6 presents a summary of what has been accomplished and discusses some directions for future research.

Chapter 2

The Argus System

Argus is an integrated programming language and run-time system [Lis88] for developing distributed programs. It is intended to be used primarily to write programs that maintain on-line data for long periods of times, such as banking systems, mail systems and airline reservation systems. This on-line data is required to remain consistent in spite of failures and concurrent access.

This chapter describes the Argus model of computation. The presentation focuses on guardians and atomic actions.

2.1 Guardians

The logical nodes in an Argus system are called *guardians.* A guardian is a special object that resides at a single physical node in a distributed system. Logically, a guardian is an abstract data object that encapsulates its resources and provides operations to access these resources. These operations, called *handlers*, can be used by other guardians to access and modify the resources controlled by the guardian. For example, consider a guardian that models a bank account (see Figure 2.1). It provides handlers to deposit and withdraw money, and to conduct balance inquiries. Another guardian might maintain a database of bibliographic references for a research group, and provide handlers to make queries and add new bibliographic entries.

Internally, a guardian contains data objects that represent the resources it is controlling. These objects can be accessed only by invoking the guardian's handlers using a remote procedure call mechanism [Nel81]. The caller sends the name of the handler to be invoked, and some arguments, to the guardian in a *call message*. When the handler invocation is finished, the results are passed back to the caller in a *return message* (see Figure 2.2). The Argus run-time

```
bank_account = guardian
    % Maintain a bank account
    deposit = handler (amount: int)
        % Deposit amount to the account
        end deposit

    withdraw = handler (amount: int) signals (insufficient_funds)
        % Withdraw amount from the account.
        % Signal error if there are not enough funds in the account.
        end withdraw

    balance = handler () returns (int)
        % Return account balance.
        end balance

    end bank_account
```

Figure 2.1: Bank account guardian.

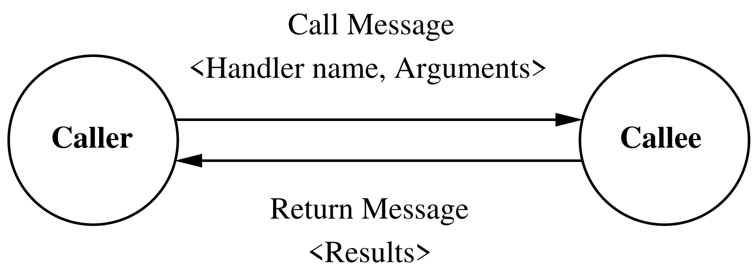

Figure 2.2: Decomposition of a handler call into a call message and a return message.

system takes care of all details of message construction and transmission inherent in the remote call mechanism.

Inside a guardian, there are one or more processes (or threads of control) that execute concurrently. Whenever a handler call comes in, a new process is created to execute the call. When the call is finished, the process sends the results back to the caller, and then is destroyed. In addition, there may be background processes that carry out tasks unrelated to a particular handler call. For example, a guardian that represents a bank branch may have a process that periodically audits all the accounts and reports any problems to the branch manager.

2.1.1 Failure and Recovery

A guardian is resilient to node failures. After a physical node recovers from a crash, the guardians that reside on the node can reconstruct their data objects by reading them from stable storage. During normal operation, the guardian's data objects are periodically copied to stable storage to minimize the loss of information in case of a node failure. The exact point at which these objects are written to stable storage is described in Section 2.3.1.

A crash destroys all processes and objects at a guardian. After the crash, the Argus run-time system recovers the guardian's objects from stable storage. The loss of processes is masked by running computations as atomic actions [GLPT76], or *actions* for short. Actions also solve the problems created by allowing concurrency within one guardian.

2.2 Atomic Actions

Two useful properties of atomic actions that help solve the problems created by concurrency and failures are *serializability* and *totality*. The serializability property implies that the effect of running a number of actions concurrently is the same as the effect of running them sequentially in some unspecified order. Serializability permits concurrent execution yet ensures that the concurrent actions do not interfere with each other. The totality of actions implies that an action either completes entirely or has no visible effect on the system's state. If an action completes, it is said to *commit*; otherwise it is said to *abort*. The effects of committed actions are made permanent by writing their changes to stable storage.

2.2.1 Atomic Objects

Atomic objects are used to implement atomic actions. Operations running on behalf of atomic actions limit their access to atomic objects. Two key properties of atomic objects help control this access to implement both serializability and totality of atomic actions.

Two Phase Locking

First, every operation carried out on any atomic object uses strict *two-phase locking* [GLPT76] to implement serializability of concurrent actions. Every operation on an atomic object is classified as either a reader or a writer. All operations that modify an object are called *writers*; other operations are called *readers*. A *read lock* is acquired before a reader accesses an object, and a *write lock* is acquired before a writer accesses an object. At any given time, at most one action can hold a write lock on an object. These locks are held until the action commits or aborts, and do not need to be re-acquired if the action references the object again. This locking scheme is called two-phase locking because the activity can be divided into two distinct phases. In the first phase, all the locks are obtained as needed. In the second phase, at commit or abort time, all the locks are released simultaneously.

Versions

Second, different versions of atomic objects are kept to implement totality. The state of an atomic object is kept in a *base* version. Whenever an operation modifies the object on behalf of an action, a *tentative* version is created to hold the modified state of the object. When the modifying action commits, the tentative version replaces the base version, and the new state of the object is written to stable storage. If the action aborts, the tentative version is thrown away.

2.2.2 Nested Actions

Atomic actions can be generalized to *nested* atomic actions by using *subactions* to build higher level actions in a hierarchical fashion, thus forming trees of nested actions [Mos81]. An action that is nested inside another is called a subaction. Non-nested actions are called *topactions*. The standard tree terminology of *parent*, *child*, *ancestor* and *descendant* applies to action trees.

Nested actions have two desirable properties. First, since siblings in an action tree can run concurrently, they allow concurrency within an action. Second, nested actions can be used as a checkpointing mechanism. For example, each handler call in Argus runs as a subaction of its caller. If a call fails for some reason, the run-time system simply aborts the call subaction and allow the caller to try again. In this way, the effects of failures are limited.

2.2.3 Locking Rules

Subactions require extensions to locking and version management. The complete set of rules is summarized in Figure 2.3. These rules ensure several things. First, a subaction is allowed to obtain a read lock on an object only if all holders of write locks on that objects are its ancestors. Similarly, a subaction can get a write lock on an object only if all holders of any sort of lock on the object are its ancestors. This prevents improper concurrent access to an atomic object because in the Argus model a child never runs concurrently with its parent.

Second, when a subaction aborts, its locks and versions are discarded and its parent action can continue from the state at which the subaction started. When a subaction commits, its locks

- 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 2.3: Rules for locking and version management for object X by subaction S.

---

and versions are inherited by its parent. If the parent aborts later, all modifications made by the subaction will be undone because the parent's locks and versions will be discarded. A stack of versions is used to implement this abort mechanism for nested actions. One version is kept for each active action that is modifying the object. When a subaction needs a new version, the version on top of the version stack is copied and the result is pushed on the stack.

2.2.4 Using Nested Actions

Subactions can be created in a number of ways.

- Every handler call runs as a subaction. This subaction is started on the caller's side and is called the *call action*. This extra action is used to ensure that handler calls have a *zero or one* semantics. If the call is successful and the called guardian replies, the call happens exactly once. If for some reason it is not possible to complete the call, the run-time system aborts the call action. The totality of atomic actions thus guarantees that effectively the call did not happen at all. Therefore running a remote procedure call as a subaction ensures that the call has clean semantics.

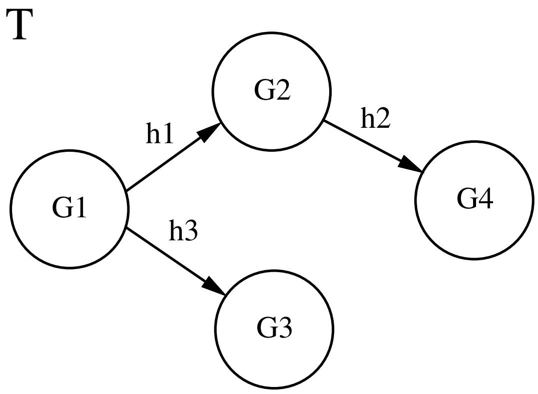

Figure 2.4: Action spreading out due to nested handler calls.

- When a handler call message is received at a guardian, a subaction is created to process it. This subaction is called the *handler action*. The handler action provides a separation between the calling and the called guardian, and ensures that each individual action runs at just one guardian.

- Explicit subactions can be created by the programmer to provide extra concurrency and a checkpointing mechanism.

2.3 Committing and Aborting Actions

A distributed program in Argus consists of a collection of guardians spread over the nodes of the network. A computation is initiated at some guardian by creating a topaction. This computation spreads to other guardians in the system by means of handler calls.

For example, in Figure 2.4, topaction T starts out at guardian G1, makes a handler call h1 to G2, which in turn makes handler call h2 to G4. When h1 is finished, T makes a final handler call h3 from G1 to G3. In this manner, topaction T, which started at G1 manages to spread to G2, G3 and G4. When T commits, all modifications made by T's descendants are written to stable storage using the standard two-phase commit protocol [Gra78]. The guardian where the topaction started acts as the coordinator for the commit protocol; guardians visited by descendants of the topaction are the participants. Information about the guardians visited by the descendants of a topaction is collected in handler call reply messages. The coordinator uses this information to compute the set of guardians that participated in the topaction. In the preceding example, G1 will be the coordinator and G1, G2, G3 and G4 will be the participants.

2.3.1 Two Phase Commit

Subaction commit and aborts are implemented using the rules in Figure 2.3. The two phase commit protocol [Gra78] is used to implement topaction commits. In the first phase, the coordinator sends out *prepare* messages to all the participants. When a participant receives a prepare message for topaction T, it releases all read locks held by T and sends a reply back to the coordinator. If the coordinator receives a reply from each participant, the second phase is started and the coordinator sends out commit messages to all the participants. When a participant receives a commit message for topaction T, it releases all write locks held on behalf T, writes the tentative versions of the atomic objects modified by T to stable storage, and installs these tentative versions as the new base versions. However, if in the first phase one of the participant refuses to prepare, or due to a failure the coordinator does not receive a reply from a participant, then the coordinator aborts the action and sends out abort messages to all the participants. On receiving an abort message, a participant releases all locks held on behalf of the action, and throws away all tentative versions of atomic objects modified by the action.

Information needs to be written to stable storage during this protocol to ensure that the effects of the committed topaction survive crashes.

1. Before a participant agrees to prepare by replying to a prepare message, it writes a *prepare record* to stable storage. This record contains the tentative versions of all atomic objects modified by the topaction. The prepare record ensures that if the coordinator decides to commit the topaction, all the modifications made by the topaction can be recovered from stable storage after a participant crash.

2. In phase two, the coordinator sends commit messages to all the participants and waits for acknowledgments. If the coordinator crashes while waiting for the acknowledgments, some participants may receive the commit message, while others may not. To solve this problem, the coordinator writes a *committing* record to stable storage before sending the commit messages. When all the commit messages have been acknowledged, the coordinator writes a *done* record to stable storage. On recovering from the crash, if there is a committing record for the topaction, but no corresponding done record, the newly recovered coordinator will resend the commit messages to all the participants and again wait for acknowledgments.

3. When a participant is notified of the commit or abort of a topaction, it records this notification on stable storage by writing either a *commit* or an *abort* record. These records are used during recovery of the guardian's state from stable storage after a crash.

If there is a commit record for a topaction, the tentative versions in the corresponding prepare record are installed as base versions. If there is an abort record for the topaction, the tentative versions are thrown away.

Various optimizations are made for topactions that do not modify anything at some participants. At such participants, no prepare record is written for topaction T. In addition, these participants are omitted from phase two of the commit protocol because they have no write locks that need to be released, or tentative versions that need to be installed. Therefore, if the topaction is read-only at all participants, the entire second phase is omitted.

Chapter 3

Viewstamped Replication

The mechanisms presented in Chapter 2 allow a programmer to construct services which are highly reliable; i.e., with a high probability, the service does not lose necessary information. However, Argus does not provide any mechanisms to make services highly available. The programmer must explicitly arrange for availability, perhaps by replicating the service onto several guardians which communicate with one another to keep their state mutually consistent. Viewstamped replication addresses this problem by automatically providing availability. Since the programming model is not changed, the programmer writes programs as before, and the run-time system is configured so that the services provided by these programs are highly available.

This chapter gives an overview of the replication scheme as described in [Oki88]. The replication scheme is then broken down into several different sections, and the implementation of each is described separately in later chapters.

3.1 Overview

Each individual guardian is replicated to obtain a *guardian group*. Each guardian group consists of several members called *cohorts*. This set of cohorts is called the guardian group's *configuration*. The configuration behaves as a single logical entity that provides the same service as the original unreplicated guardian. Each cohort has a system-wide unique name called its *guardian identifier*, or *gid* for short. Guardian groups have unique names called *group identifiers*, or *groupid* for short. Each cohort knows its own gid as well as the configuration and the groupid of the guardian group to which it belongs. It is assumed that a guardian group's configuration never changes — the set of cohorts that belong to the group is fixed at group creation time.

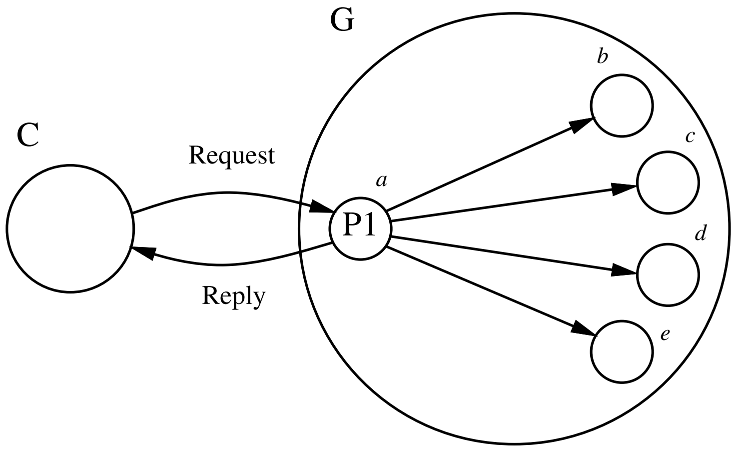

Figure 3.1: Normal operation of replicated guardian group G consisting of cohorts *a, b, c, d* and *e* with primary *a*.

One member of the configuration is designated the *primary*; the other members are *backups*. During normal operation, handler calls made to the guardian group are routed to the primary. The primary executes all incoming handler calls and propagates information about these handler calls to the backups in the background. If any cohort crashes or becomes inaccessible because of network failures, the remaining cohorts undergo a reorganization. If the original primary is no longer accessible, a new primary is selected during the reorganization.

Figure 3.1 illustrates the interaction between a guardian group and a client. G is a guardian group consisting of the five cohorts *a, b, c, d* and *e*. Cohort *a* is the primary, and the rest are backups. The five cohorts are grouped together in the picture to suggest that they form a single logical entity G, even though physically they may be distributed over a network. The client C communicates with G, and its requests are routed to the primary *a*. Cohort *a* carries out the request and replies back to C; it also sends information about completed requests to the backups in the background.

Suppose a failure causes a network to be partitioned as in Figure 3.2. In response to this failure, G reorganizes itself automatically to remain available. This reorganization is called a *view change*, and the algorithm that carries it out is called the *view change algorithm*. The remaining cohorts select a new primary, *b* in this example, since the old primary *a* is no longer accessible. After this reorganization, *b* is ready to receive new requests on behalf of G.

Consider what happens to transactions that are active during a view change. Suppose transaction T1 had made some handler calls to the old primary *a* before the view change. T1 then decides to commit, and a prepare message is sent to the new primary *b*. What should

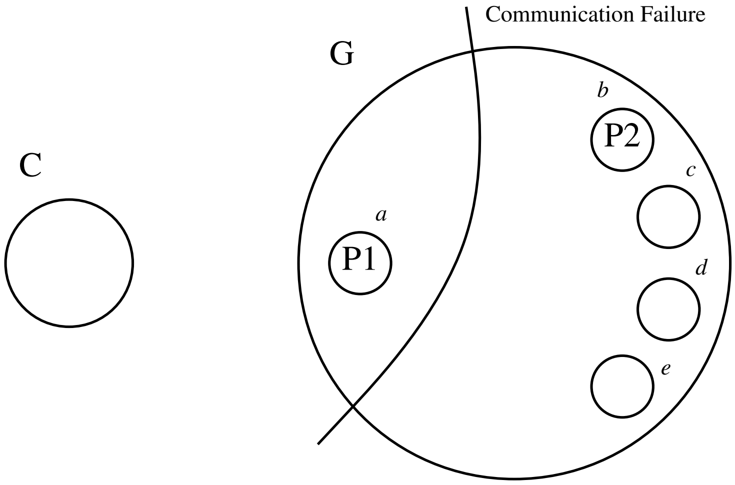

Figure 3.2: Guardian group partitioned due to network failure with old primary *a*, and new primary *b*.

---

b do on receiving this message? If all the activity carried out on behalf of T1 at *a* has been propagated to *b*, then *b* can go ahead and let T1 commit. However, if some of this information did not make it to *b* before the partition, *b* does not know about all the activity carried out at G on behalf of T1, and has to abort T1. Viewstamped replication provides an inexpensive way of determining what information is "known" at a guardian, and what information needs to be known to allow a transaction to commit.

Toes rest of this chapter explains the various parts of the viewstamped replication scheme.

3.2 View Management

Each guardian group goes through a sequence of node and communication link failures and recoveries. Since these failures can interfere with the semantics of the service provided by the guardian group, a mechanism is needed to mask these failures from the outside world. The concept of *views* provides this mechanism. A view exists for a period of time during which the communication capability of the guardian group remains unchanged. It identifies the primary and the set of cohorts that are active during this period of time.

In Figure 3.3, guardian group G starts out in view $\{a : b, c, d, e\}$; *a* is the primary, and *b*, *c*, *d* and *e* are the backups. Because of a communication failure, which makes *a* unreachable from the rest of the group, G then undergoes a view change and enters the new view $\{b : c, d, e\}$ with *b* as the new primary.

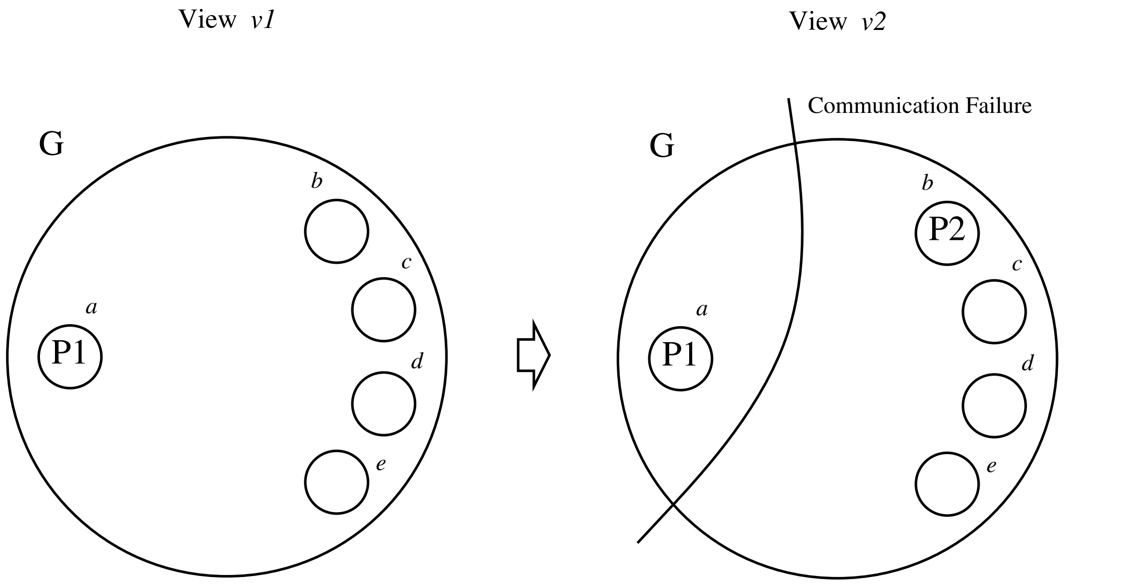

Figure 3.3: View change in guardian group G due to partitioning of old primary *a* from the rest of the cohorts.

Each view is named from a set of totally ordered *viewids*. The scheme used for generating the viewids must guarantee that they are unique, and that viewids for later views should be larger than viewids for earlier views.

When the communication capability of the guardian group changes, due to either failure or recovery of a communication link or a node, a *view change algorithm* is initiated to form a new view. A modification of the virtual partitions protocol proposed in [ESC85] is used for this purpose. The algorithm creates a new view consisting of at least a majority of the cohorts in the configuration. It also assigns a unique *viewid* to this new view.

In Figure 3.3, a partition in the network changes the communication capability of the guardian group G. The view change algorithm discards the old view $\{a : b, c, d, e\}$ and creates the new view $\{b : c, d, e\}$ with new primary b. It also assigns the unique viewid $v2$ to the new view.

The view change algorithm also finds the most “up-to-date” state from the local states of all the members of the old view that are accessible in the new view. Since each view consists of at least a majority of the cohorts in the configuration, the intersection of the old and the new view must be non-empty, and at least one such state is always available. This state is transmitted to the new primary as part of the view change and is used as the initial state of the guardian in the new view.

3.3 Mechanisms for Running Transactions

The system makes several guarantees to correctly implement atomic transactions. First, if a topaction commits in some view, its effects are guaranteed to survive all subsequent view changes by waiting for these effects to reach at least a majority of the cohorts in the configuration before allowing the topaction to commit. When the next view is formed, its initial state will contain all the effects of the committed topaction. Since this initial state is propagated to at least a majority of the cohorts at view change time, the effects of the committed topaction will survive the next view change too. By induction, the effects of the committed topaction will be known in all future views.

Second, if information about a committed subaction S is lost during a view change, then the topaction T that is an ancestor of S must not be allowed to commit. This is done by allowing T to prepare at G only if all effects of T's descendants are "known" at G's primary. If any of these effects have been lost during intervening view changes, T must be aborted. *Timestamps* are used to determine what is "known" at a given cohort.

3.3.1 Timestamps

Any activity at the primary which needs to be propagated to the backups is called an *event*. Completion of handler calls, topaction commits and topaction aborts are examples of events. *Timestamps* are assigned to each event; these timestamps must be unique within a view. The timestamps should also form a totally ordered set within a view, with later events receiving larger timestamps than earlier events.

The replication scheme requires that information about events be propagated to the backups in timestamp order. This guarantees that if a cohort knows about an event, then it knows about all earlier events in the same view.

3.3.2 Viewstamps

Timestamps are tagged with the viewid of the view they were generated in to form *viewstamps*. Since viewids are unique across a guardian group, and timestamps are unique within a view, a viewstamp uniquely names an event that happened at the guardian group. Therefore, we can use a set of viewstamps S to describe what a cohort "knows". This set is called the *viewstamp history* of the cohort.

For example, suppose a cohort knows about events with timestamps 1 through 4 from view

$v1$, and about events with timestamps 1 and 2 from view $v2$. Then, its viewstamp history is

$$\{\langle v1.1\rangle, \langle v1.2\rangle, \langle v1.3\rangle, \langle v1.4\rangle, \langle v2.1\rangle, \langle v2.2\rangle\}$$

Since viewids are totally ordered, and all timestamps within a view are totally ordered, a total order can also be imposed on viewstamps. Viewstamps within the same view are ordered by timestamps. Viewstamps in different views are ordered by viewids. More rigorously, we can define a total order $<$ on viewstamps such that $\langle v1.t1\rangle < \langle v2.t2\rangle$ if

$$(v1 < v2) \text{ or } ((v1 = v2) \text{ and } (t1 < t2))$$

The next section describes how these mechanisms are used for topaction commits.

3.4 Transaction Processing

In a transaction processing system, clients make handler calls to servers, and act as two-phase commit coordinators at commit time.

3.4.1 Handler Calls

Clients make handler calls to servers. They send their belief about about the server's current viewid along with other relevant information to the server primary in the call message. If the viewid in the call message does not match the view the primary is in, the primary rejects the call message since it may lack results from earlier calls done by T, and the client tries again with another viewid. Otherwise, the handler call is processed at the primary. The primary generates a new viewstamp for the handler call event. This viewstamp flows back on the reply message to the client. For each topaction, the client keeps track of a set of viewstamps $VSS(T, S)$ for each server S that the topaction T visits.

In Figure 3.4, the client sends a call message to the server G, which is in view $v2$. When G's primary P receives the call message, it generates a new viewstamp $\langle v2.4\rangle$ for the handler call event, does the processing required by the handler call, adds the new viewstamp to its viewstamp history, and sends the viewstamp back to the client. The client adds $\langle v2.4\rangle$ to $VSS(T, G)$.

3.4.2 Two Phase Commit

As described above, the coordinator keeps a set of viewstamps $VSS(T, S)$ for each server S that participated in the topaction T. At commit time for topaction T, the coordinator sends a

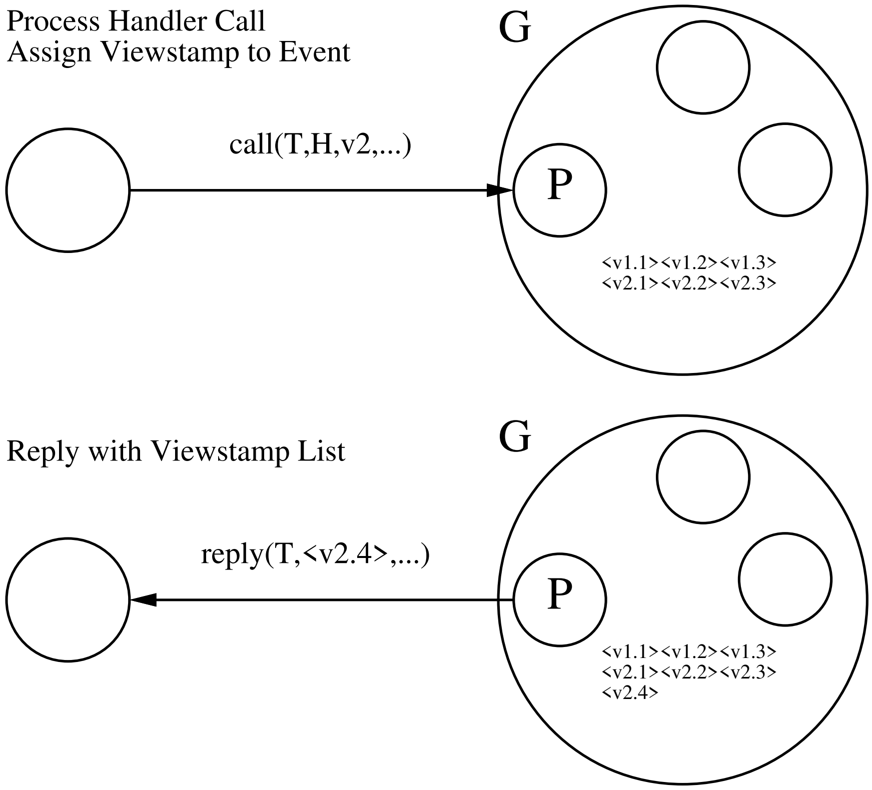

Figure 3.4: Handler call processing. Viewids flow from the client to the server, and viewstamps flow from the server to the client.

---

prepare message to each participant. The prepare message sent to server S contains the list of viewstamps $VSS(T, S)$. The server primary now needs to decide if enough information is known to allow the topaction to commit. Each cohort C's viewstamp history $VSH(C)$ describes the events known by C; see Section 3.3.2. Therefore, the primary P knows enough information to allow T to commit if

1. $VSS(T, S) \subseteq VSH(P)$
2. For all transactions X, if T depends on X then $VSS(X, S) \subseteq VSH(P)$.

The first condition is easy to check. A property of $VSH(P)$ removes the need for checking the second condition. Let T depend on transaction X. The serializability property of transactions guarantees that X has already committed. Since information about committed transactions is guaranteed to survive into all subsequent views, $VSS(X, S) \subseteq VSH(P)$. Therefore, the second condition is always guaranteed to hold and we do not need to check for it explicitly.

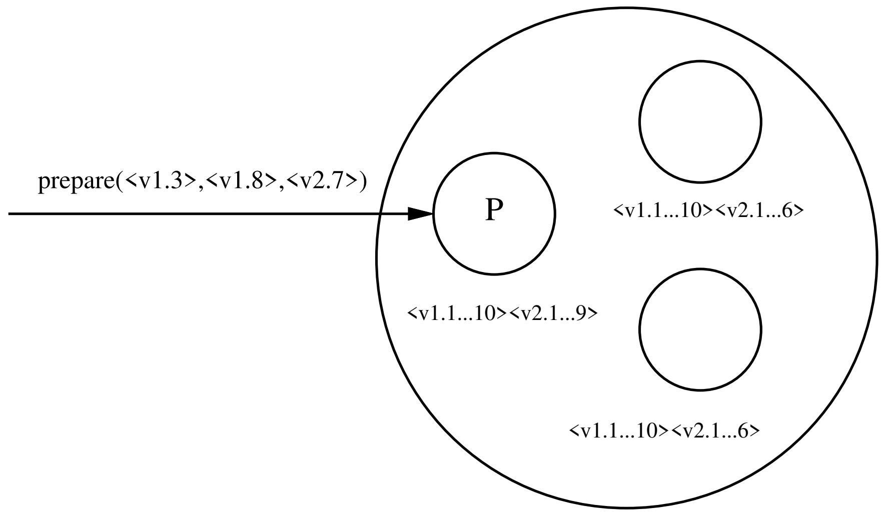

Figure 3.5: Deciding whether a server should let a topaction commit.

In Figure 3.5, the client sends the viewstamp set $\{\langle v1.3\rangle,\langle v1.8\rangle,\langle v2.7\rangle\}$ to the server G. Since the primary's viewstamp history is a superset of this set, the primary knows that enough information is available to allow the topaction to commit.

However, this is not sufficient. To make commits permanent, we must also ensure that a majority of the cohorts have enough information. For example, suppose we allow the topaction to commit after checking only that the primary has enough information. Right after the commit, the primary could crash, and nobody in the resulting view would have all the information about the topaction — even though it committed. Therefore, before the primary can allow the topaction to commit, it has to ensure that a majority of the cohorts in the configuration know about all the activity done on behalf of the topaction. It does this by waiting for the required information to propagate to at least a *sub-majority* of the cohorts in the configuration, where a sub-majority is one less than a simple majority; a sub-majority plus the primary constitutes a majority.

Consider Figure 3.5 again. The primary has to wait until at least one of its backups hears about the event with viewstamp $\langle v2.7\rangle$, so that a majority of the cohorts in the configuration will know everything done on behalf of the topaction. Waiting for the required information to propagate to a sub-majority corresponds to writing the prepare record to stable storage in the unreplicated transaction system.

Chapter 4

Transaction Implementation

This chapter describes my implementation of transactions for viewstamped replication. The Argus implementation was used as the basis for this work. First, I give an overview of the portion of the run-time system responsible for transaction processing. Then, in Section 4.2 the implementation of nested actions is described. In Section 4.3 the mechanism for propagating information from the primary to the backups is described. Section 4.4 describes the mechanism for locating the current primary of a guardian group. Section 4.5 describes the implementation of handler calls. Section 4.6 describes the two phase commit protocol used for topaction commits. Section 4.7 presents some performance figures for transactions running under viewstamped replication. Finally, Section 4.7.3 compares the performance of the replicated system with the Isis replicated transaction facility [Bir85].

4.1 Overview

The organization of the transaction processing run-time system is given in Figure 4.1. The primary executes incoming handler calls and participates in the two phase commit protocol with primaries of other guardian groups. Both handler call processing, and the commit protocol generate *events* that are propagated to the backups by several *Sender* processes. Each sender process is responsible for sending event records to one backup; it communicates with the *Receiver* process at that backup to achieve reliable and in-order transmission of the event records.

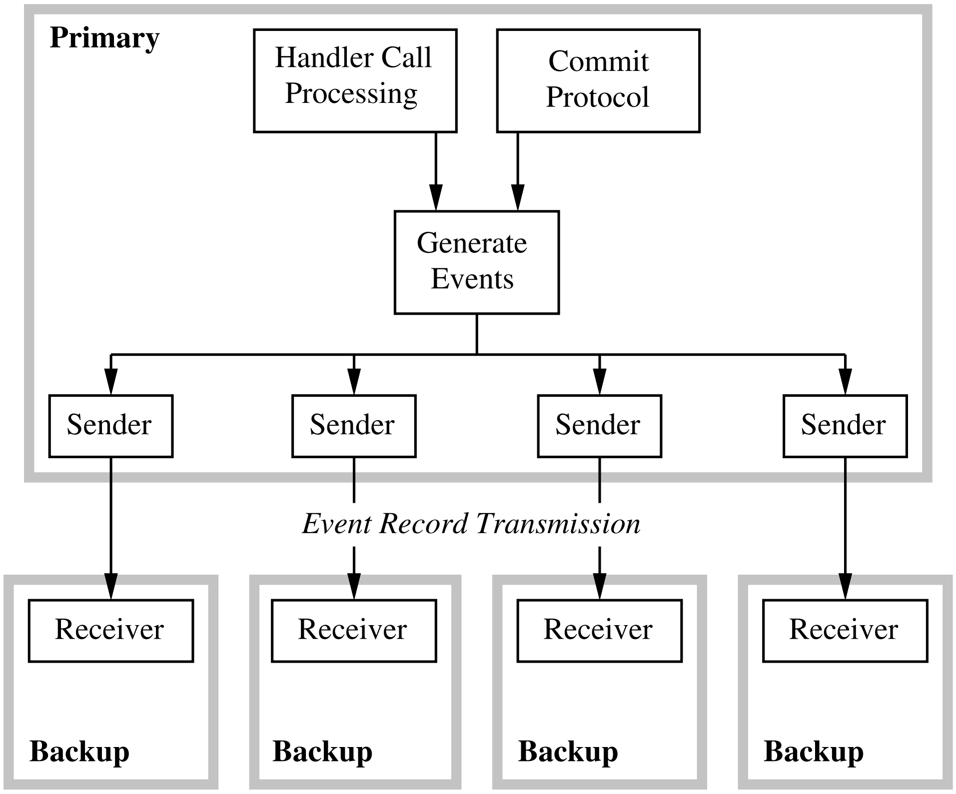

Figure 4.1: An overview of the transaction processing implementation.

4.2 Actions

This section describes the implementation of atomic actions for viewstamped replication. Section 4.2.1 describes a useful method for visualizing nested actions and presents an example that will be used in the rest of this section to illustrate different points. Section 4.2.2 describes a mechanism for naming nested actions that simplifies the implementation of some operations on nested actions. Section 4.2.3, Section 4.2.4 and Section 4.2.5 describe information about nested actions that is collected and then used in the implementation of two phase commit.

4.2.1 Action Trees

Nested transactions can fan out to several guardian groups via handler calls. Actions trees provide a useful way to visualize the state of a transaction that spans several guardian groups. The tree is a simple model for the nested structure of a transaction. Each action in the transaction maps to a node in the tree. The nodes in the tree can be marked one of three ways — active,

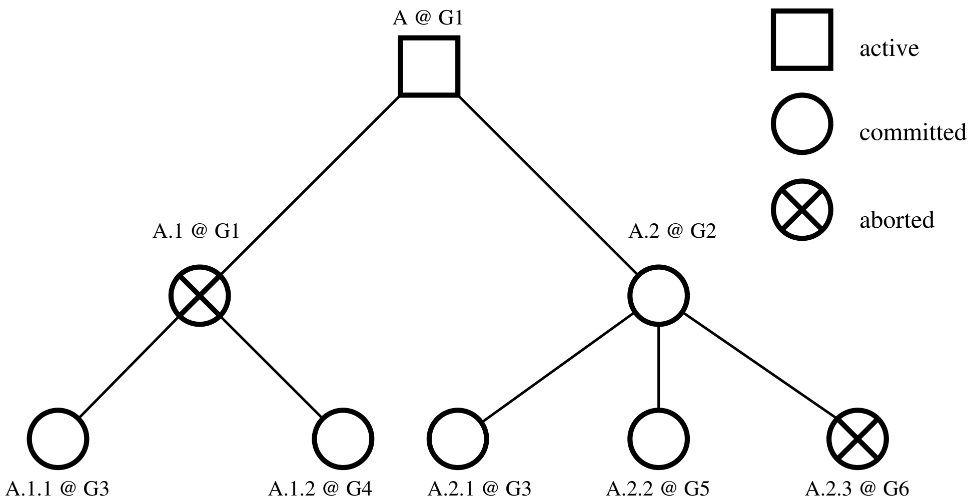

Figure 4.2: Action tree showing the location and status of topaction A.

aborted or committed — to show the state of the corresponding action. In addition, the location at which an action runs is also marked on the action tree.

Consider Figure 4.2. Action A is active at guardian group G1. Subaction A.1 at G1 aborted. A.1's children, A.1.1 at guardian group G3 and A.1.2 at guardian group G4 have both committed; however, since their parent A.1 has aborted, both A.1.1 and A.1.2 are considered to have aborted with respect to the topaction A. Subaction A.2 ran at guardian group G2 and has committed. Two of its children, A.2.1 at G3 and A.2.2 at G5 have committed. The remaining child, A.2.3, ran at G6 and has aborted.

Recall that a handler call creates two actions: a *call action* at the caller, and a *handler action* at the callee. For simplicity, only handler actions are shown in this action tree. The real action tree can be obtained by inserting appropriate call action nodes above the handler action nodes in Figure 4.2.

4.2.2 Action Identifiers

An actions is named by an action identifier (*aid* for short). The *aid* for an action A is just the *aid* for its parent action concatenated with an extra entry. This extra entry is of the form $\langle$*uid*, *groupid*$\rangle$ where *groupid* is the identifier of the guardian group where A ran, and *uid* is a small tag to differentiate between siblings that run at the same guardian group. This naming scheme is just a variation on the labels for the actions in Figure 4.2. Action A's *aid* is [$\langle$1,G1$\rangle$], action A.1's *aid* is [$\langle$1,G1$\rangle$,$\langle$1,G1$\rangle$] and action A.1.2's *aid* is [$\langle$1,G1$\rangle$,$\langle$1,G1$\rangle$,$\langle$2,G4$\rangle$].

Given this representation for an *aid*, it is easy to implement the following operations on action identifiers.

- check whether an action is an ancestor of another action.
- find an action's parent.
- find the *groupid* of the guardian group where an action ran.
- find the least common ancestor of two actions.

4.2.3 Participant Set

At topaction commit time, the coordinator needs to know the guardian groups at which the topaction ran. This is done by collecting a participant set up the action tree. When an action starts running at a particular guardian group G, its participant set is initialized to {G}. If an action commits, its participant set is merged into its parent's participant set; if the action is a handler action, the participant set is passed back to the parent call action by piggybacking it on the reply message for the handler call. In Figure 4.2, A.2's participant set is {G2, G3, G5}, and A's participant set is {G1, G2, G3, G5}. The locations of aborted subactions are not included in the participant set. Therefore, G4 and G6 will not be involved in the topaction commit protocol for A.

4.2.4 Aborts Set

The aborts set for an action X is the set of subactions of X that have aborted. For example, the aborts set for action A in Figure 4.2 is

$$\{\text{A.1, A.1.1, A.1.2, A.2.3}\}$$

These sets are computed up the action tree just as the participant sets are. There is a more compact representation for these sets. If an action A is a member of the aborts set, then all proper descendants of A are removed from the aborts set. This compaction is correct because A's membership in the aborts set implies the membership of all its descendants (descendants of an aborted action are also considered aborted). The aborts set given above shrinks to {A.1, A.2.3} after applying this compaction.

4.2.5 Viewstamp Sets

As discussed in Section 3.4.2, the prepare message for topaction T sent to participant P contains the viewstamp set $VSS(T,P)$. These sets are also computed up the action tree. If action A has n participants, it will have n viewstamp sets, one per participant. It may seem expensive to move these n sets around, but a simple optimization drastically reduces the size of these sets.

In Section 3.3.1, it was noted that if a cohort knows about an event with a particular timestamp in view v, then it knows about all events in view v that have smaller timestamps. More formally, if $\langle v.y \rangle$ is in $VSH(C)$ and $y > x$ then $\langle v.x \rangle$ is in $VSH(C)$. Therefore, if $VSS(T,P)$ contains more than one viewstamp for a particular view, than all but the largest of these viewstamps can be safely discarded. Let us see why this is a correct optimization to make. Let participant P's primary be X. Let $\langle v.ts \rangle$ be the largest viewstamp in $VSS(T,P)$ for view v. If $\langle v.ts \rangle \in VSH(X)$, then all the discarded viewstamps will also be in $VSH(X)$. If $\langle v.ts \rangle \notin VSH(X)$, then $VSS(T,P)$ is not a subset of $VSH(X)$. Therefore, the reduced viewstamp set is a subset of $VSH(X)$ iff the original viewstamp set $VSS(T,P)$ is a subset of $VSH(X)$, and the optimization is correct.

After making this optimization, there will be at most one viewstamp per view in this participant's viewstamp set. For example, let $VSS(T,P)$ for an action be

$$\{\langle v1.3 \rangle, \langle v1.5 \rangle, \langle v1.6 \rangle, \langle v2.1 \rangle, \langle v2.8 \rangle\}$$

The reduced viewstamp set will be just $\{\langle v1.6 \rangle, \langle v2.8 \rangle\}$. This optimization is performed as the viewstamp sets are merged up the action tree. The optimization was first given in [Oki88], but it was used directly, and not presented as a method for reducing the size of the viewstamp sets.

4.3 Event Log

When a backup joins a view, the primary may need to update the backup's state so that it corresponds to the primary's state. One way to perform this update is to send the primary's whole state over the network. This method may be unnecessarily expensive if the differences between the primary and the backups' states is small. A more efficient method that transmits just the differences between the two states exists. The key insight here is that a cohort's state can be represented by the sequence of events that it knows about. Therefore, if each cohort stores the events it knows about in an *event log*, the primary can efficiently update a backup's state by sending the differences between the primary's and the backup's log. Log compaction techniques can be used to prevent these logs from growing without bound. My implementation uses event

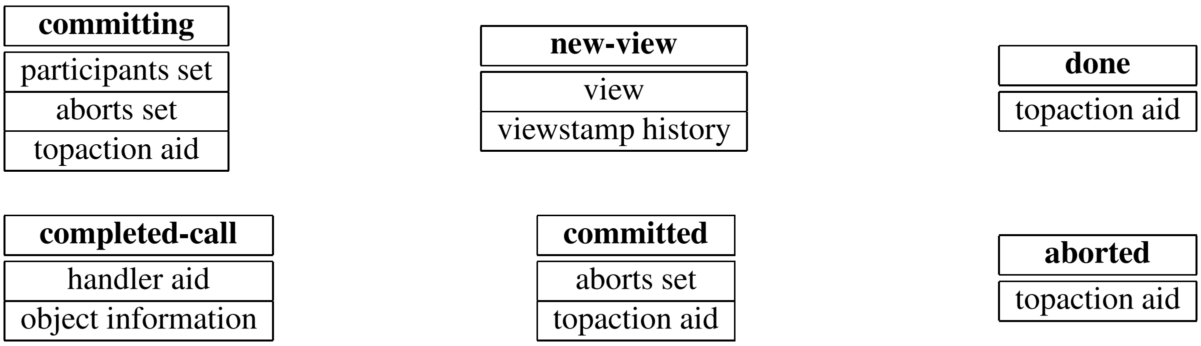

Figure 4.3: Format of event records

logs to transfer information from the primary to the backups. However, the implementation does not contain a log compaction scheme.

4.3.1 Event Types

There are six different kinds of events that can be stored in the event log. Their formats are shown in Figure 4.3. *Committing* and *done* events are generated at the coordinator of a committing action. A *completed-call* event is generated when a handler call completes at a server. The format of completed-call event records is described later in Section 4.5. *Committed* and *aborted* events are generated when a participant is notified of the status of a topaction. *New-view* records are written to the log by the newly selected primary as part of the view change process.

4.3.2 Viewstamp History

As described in Section 3.3.1, a cohort's viewstamp history is the set of viewstamps that describe what the cohort knows; i.e., a cohort's viewstamp history is the set of viewstamps for all the events in the cohort's event log. The technique used to compress the viewstamp sets generated during transaction processing, as described in Section 4.2.5, is also used to get a compact representation of a cohort's viewstamp history. If a cohort has an event from view v in its event log, then it has all other events with smaller viewstamps from view v. This implies that if $\langle v.x \rangle \in VSH(C)$ and $y < x$ then $\langle v.y \rangle \in VSH(C)$. Therefore, the representation of a cohort viewstamp history just records the largest viewstamp from each view about which view

```
extent[b]:    viewstamp  % Viewstamp of last event known to be in b's log
flood_to[b]: viewstamp  % Viewstamp of event we want in b's log

sender[b] = process
   % Use streaming protocol to reliably send the portion of the event log between
   % extent[b] and flood_to[b] to b.
   while true do
        wait for extent[b] < flood_to[b]
        % Invoke streaming protocol
        ...
        end
   end sender[b]
```

Figure 4.4: Sending the event log to a backup

the cohort knows. For example, let a cohort's viewstamp history be

$$\{\langle v1.1\rangle, \ldots \langle v1.29\rangle, \langle v2.1\rangle, \ldots \langle v2.11\rangle\}$$

This history is uniquely represented by the set $\{\langle v1.29\rangle, \langle v2.11\rangle\}$.

4.3.3 Primary to Backup Transmission

At several points, the primary must guarantee that some information has reached a majority of the cohorts in the configuration. For example, when a participant primary receives a prepare message for a topaction T, it needs to ensure that all the completed-call events for topaction T have reached a majority of the cohorts in the configuration. This ensures that if T commits, its effects will survive subsequent view changes. The primary guarantees that particular events are known at a majority of the cohorts by sending portions of the event log to its backups.

The mechanism used to transmit a portion of the event log from the primary to a backup *b* is illustrated in Figure 4.4. *extent*[*b*] is the viewstamp of the last event known by the primary to be in *b*'s event log. *flood-to*[*b*] is the viewstamp of the most recent event in the portion of the log that has to be sent to *b*. The process *sender*[*b*] waits until some event needs to be sent to be *b*. It then uses a reliable message delivery system to send the portion of the event log between *extent*[*b*] and *flood-to*[*b*] to *b*. The current implementation uses a sliding window protocol [Tan81] for fast and efficient message delivery. This protocol updates *extent*[*b*] based on the information present in acknowledgment messages received from *b*.

The interface to this transmission mechanism is provided by the procedure *force-events* shown in Figure 4.5. *Submajority(backups)* returns a subset of the backups that forms a

```
force−events = proc (events: set[event])
    max_vs: viewstamp := max {e.vs | e ∈ events}
    dests: set[cohort] := submajority(backups)
    for b: cohort in dests do
        flood_to[b] := max {flood_to[b], max_vs}
        end
    % Wait until max_vs reaches a submajority
    wait for ⋀_{b∈dests}(extent[b] ≥ max_vs)
    end force−events
```

Figure 4.5: Forcing a set of events to a majority of the configuration

sub-majority of the configuration. The primary and a sub-majority together form a majority in the configuration. Therefore, to guarantee that a set of events is known at a majority, the primary sends the relevant portion of the event log to a sub-majority of its backups[1] and waits until it receives acknowledgments.

4.4 Locating Primaries

Handler call and prepare messages need to be sent to the destination guardian group's primary. These messages also need to contain the correct viewids so that old primaries safely ignore the message. To achieve this, each guardian maintains a cache that maps guardian group identifiers to viewids and primary identifiers. This cache may be out of date because of view changes that the guardian has not heard about. When sending a handler call or a prepare message, the primary identifier and viewid are looked up in this cache and the message is sent.

If the cohort that receives the message detects that the information retrieved from the cache is out of date — the viewid in the message is not correct, or the cohort is not the current primary — the cohort replies with a message containing the correct information. This information is used to update the cache and the message is sent again.

4.5 Handler Calls

The client primary uses the mechanism described in Section 4.4 to send a *handler call* message to the server primary. The server primary creates a handler action that runs the code associated

[1] In the current implementation, one sub-majority is picked at the start of a view and information is always forced to this sub-majority for the duration of the view.

```
% Type of completed call event record
type completed_call_record = record [
    id:      aid,               % Action identifier
    objects: array[obj_info]    % Locks and tentative versions
    ]

% Type of information kept for each accessed object
type obj_info = oneof [
    read_lock:   object_id,   % Read locked object's id
    write_lock:  write_info   % Write locked object info
    ]

% Type of information kept for write locked objects
type write_info = record [
    id:        object_id,  % Object identifier
    tentative: object      % Tentative version
    ]
```

Figure 4.6: Format of completed call event record

with the handler specified in the handler call message. Running this code may involve nested handler calls to other guardians; see Figure 2.4 for an example. After running this code, the handler action generates a *completed-call* event record for this handler call.

The format for this event record is given in Figure 4.6. The completed call record contains a list of all the locks acquired by the handler action. In addition, the record contains the tentative versions of all objects modified by the handler action. The completed call record is appended to the event log, and a new viewstamp is generated for it. This viewstamp is merged into the viewstamp set collected for the handler action as described in Section 4.2.5. Finally, the handler action commits by sending a reply message to the client. This reply message contains the viewstamp, participant and aborts set collected for the handler action as well as the results of executing the actual handler named in the handler call message sent by the client. If the handler action aborts, a reply is sent back to the client indicating that the handler action aborted. No other information is sent to the client in this case.

4.6 Two Phase Commit

Sections 4.6.1 and 4.6.2 describe the implementation of two phase commit in the replicated transaction system. T refers to the topaction being committed. The primary of the guardian group that created T is the commit coordinator.

4.6.1 Phase One

The coordinator sends a prepare message to each participant's primary The prepare message sent to participant P's primary has three fields —

1. T's action identifier.

2. $VSS(T, P)$. The participant primary uses this set to determine whether or not T should be allowed to commit.

3. List of action identifiers of aborted subactions of T. The participant primary releases locks and discards tentative versions held on behalf of these aborted subactions.

All the information required to generate the prepare messages is collected by the mechanisms described in Section 4.2. The coordinator waits for replies to all the prepare messages. If a reply is not received from a participant in a certain amount of time, the coordinator re-sends the prepare message. If no reply is received after the prepare message has been re-sent a certain number of times, the participant is assumed to be unreachable, and the topaction is aborted. If all the participants agree to prepare by sending a *prepare-ok* message as a reply, the coordinator proceeds to phase two. If some participant refuses to prepare by sending a *refuse-prepare* message, the topaction is aborted.

When participant P's primary X receives a prepare message, it extracts the viewstamp set $VSS(T, P)$ from the message. If $VSS(T, P)$ is a subset of the primary's viewstamp history $VSH(X)$, enough information is present to allow the topaction to commit; if not, the primary sends a *refuse-prepare* message to the coordinator. If the topaction can commit, the primary forces the set of events corresponding to the viewstamp set $VSS(T, P)$ to a sub-majority using the procedure *force-events* described in Section 4.3.3. This forcing is analogous to writing the prepare record to stable storage in the unreplicated transaction system. The write to stable storage in an unreplicated system ensures that the modifications made by T will survive subsequent crashes. The forcing of $VSS(T, P)$ to a sub-majority in the replicated system ensures that the modifications made by T will survive subsequent view changes. Finally, X sends a *prepare ok* message to the coordinator.

4.6.2 Phase Two

The coordinator appends a *committing* event record to the event log and then forces it to a sub-majority. If the coordinator crashes during phase two, this record is used to determine whether or not T committed. If the *committing* record survives into the new view, T is assumed to have committed; otherwise, T is aborted. The coordinator forces the *committing* event to a sub-majority; when the force is finished, the *committing* event is guaranteed to survive into subsequent views, and the topaction T is known to have committed. The coordinator then sends *commit* messages to all the participant primaries. When these commit messages have all been acknowledged, a *done* event record is appended to the log and lazily propagated to the backups. The *committing* and *done* records serve the same purpose as the corresponding records written to stable storage in the unreplicated system; see Section 2.3.1.

When the participant primary receives a *commit* message from the coordinator, a *commit* event record is appended to the event log and forced to a sub-majority. When the force has finished, an acknowledgment is sent to the coordinator.

4.6.3 Aborts

The coordinator may decide to abort the topaction (for example, if a participant refuses to prepare or if a participant is unreachable). The coordinator then generates an *aborted* record for the topaction and appends it to the event log. It also sends messages to all the participants informing them of the topaction abort so that they may release locks held by descendants of this topaction. The coordinator does not make sure that these messages reach the participants, because if some other action wants to acquire a lock held by a descendant of this topaction, it can always find out about the abort by contacting the coordinator.

4.6.4 Optimizations

Several simple optimizations were made to the two phase commit protocol to improve system performance and decrease the latency of the commit protocol.

1. Since the coordinator is also a participant, the coordinator sends messages to itself. This can be easily optimized so that instead of sending a message to itself, the coordinator just performs the computation that would be done in response to the message.

2. The outcome of the topaction (commit vs. abort) is known once the *committing* record has been forced to a sub-majority at the coordinator. Therefore, the user code can be allowed to continue at this point, and the rest of phase two can be finished in the background by the run-time system. This decreases the latency of the two phase commit protocol from the point of view of user code.

3. The optimizations made in the unreplicated system for topactions that were read-only at some participants can also be applied here.

If T is read-only at participant P, then P is not included in phase two of the commit protocol. When P's primary gets the *prepare* message in phase one, after forcing $VSS(T, P)$ to a sub-majority, it appends a *commit* event record to the event log and sends a reply to the coordinator indicating that it is prepared read-only. The participant P does not need to be included in the rest of the commit protocol. If all the participants are read-only, during phase two the coordinator does not generate either a *committing* or a *done* record because there are no participants that need to be informed of T's commit.

In the unreplicated system, no prepare record was written to stable storage at a participant where the topaction was read-only. In the replicated system, we still require the events associated with $VSS(T, P)$ to be forced to a sub-majority during phase one. This is done because some failure and recovery sequences can result in multiple concurrent primaries, where the primary in the most recent view does not know all the completed call events generated by T whereas some out of date primary has all these events in its log. If the prepare message arrives at the out of date primary, the topaction should not be allowed to commit because another action might have acquired conflicting locks at the other primary. Attempting to force $VSS(T, P)$ to a sub-majority ensures that this condition is detected, because at least a majority of the cohorts in the configuration will know that the primary that received the prepare message is out of date. These cohorts will not accept the portions of the event log that the primary sends to them. Therefore, the force will not succeed, and the topaction will be aborted.

4.7 Performance

This section discusses the performance of the replicated transaction system and compares it with the original unreplicated system. Both systems are very similar. In particular, the replicated system was implemented by adding the replication scheme to the unreplicated Argus system. If future technology speeds up the unreplicated system, the same increase in speed should show up in the replicated system.

The performance figures were collected by repeating the experiments a large number of times and dividing the elapsed time by the number of repetitions. This technique was motivated by the very large granularity (10 ms) of the system clock. All measurements were obtained on MicroVAX II's running the 4.3 BSD operating system and connected by a 10 megabits/second ethernet. For more detailed performance measurements of the unreplicated system see [LCJS87].

Handler Call	Orig-inal	Replicated			
		1,1	1,3	3,1	3,3
Null	19	20	20	20	20
Read	23	24	24	24	25
Write	23	24	25	25	25

Table 4.1: The time (in milliseconds) required to make a handler call. Original refers to the unreplicated system. a, b refers to a replicated system with a replicas for the client and b replicas for the server.

4.7.1 Handler Calls

We measured the times for three different kinds of handler calls. Null handler calls have no user code to execute. Read handler calls read a small object, thereby acquiring a read lock. Write handler calls modify a small object, thereby acquiring a write lock and creating a tentative version. These calls were run on both the original system, and the replicated system with varying degrees of replication. The times for these calls are given in Table 4.1. The column labeled "Original" gives handler call times on the unreplicated system. The other columns give the times for the replicated system with different degrees of replication. For example, the column labeled $1, 3$ gives handler call performance figures for a system where the client guardian group is composed of one cohort and the server guardian group is composed of three cohorts.

In all of the cases covered, the difference between the unreplicated and the replicated handler calls is at most two milliseconds. This difference is due to several factors:

1. Primary and viewid lookups at the caller.

2. Viewid checks at the callee.

3. The viewstamp sets flowing on the reply messages.

4. The generation of completed call records.

5. Contention for processing resources with the processes that send the event log to the backups.

Top-action	Orig-inal	Replicated			
		1,1	1,3	3,1	3,3
Read	40	43	61	43	63
Write	127	50	62	70	82

Table 4.2: The time (in milliseconds) required to run a topaction consisting of one handler call. Original refers to the unreplicated system. a, b refers to a replicated system with a replicas of the client and b replicas of the server.

4.7.2 Top Actions

The read and write handler calls described in the previous section were also used to collect performance measurements for topactions. Table 4.2 gives performance figures for topactions that consist of one handler call each. A read topaction consists of one read handler call; a write topaction consists of one write handler call. In the following discussion, O will refer to an unreplicated system and $R(a, b)$ will refer to a replicated system with a cohorts for the client guardian group and b cohorts for the server guardian group.

Read Topactions

Let us examine the numbers for read topactions. In O, a read topaction consists of a read handler call and phase one of the two phase commit protocol; the second phase is optimized away. The handler call takes 23 ms, and the message round trip delay involved in sending a prepare message and waiting for a reply from the participant accounts for the remaining 17 ms.

In $R(1, 1)$ the handler call takes 24 ms. The remaining 19 ms are spent in phase one of the two phase commit protocol. Phase one of the replicated system involves forcing a portion of the event log to a sub-majority at the participant. However, in $R(1, 1)$ a sub-majority has size zero, so no extra time is spent forcing the event log. Therefore, a read topaction takes roughly the same time in both O and $R(1, 1)$.

In $R(1, 3)$ however, a sub-majority has size one. Therefore, in $R(1, 3)$ there is a message round trip delay while the server primary forces a portion of the event log to a sub-majority during phase one. This accounts for the extra 18 ms required when the server guardian group is replicated.

In $R(3, 1)$, the coordinator for two phase commit is replicated. However, the coordinator for a read-only topaction does not force anything to a sub-majority. Therefore, no extra overhead is involved in replicating the coordinator, and the time for $R(3, 1)$ is the same as the time for

$R(1,1)$. For the same reason, the difference between $R(1,3)$ and $R(3,3)$ is negligible for a read topaction.

Write Topactions

In the original system O, the time required to run a write topaction is 127 ms, which is more than three times as much as the time required to run a read topaction. This extra time is spent writing different records to stable storage. However, the implementation of O for which these performance measurements were collected does not use real stable storage. It fakes stable storage by writing to a disk. In a system that uses stable storage, the time to run a write topaction will be even greater.

In the replicated system, forcing to a sub-majority takes the place of writing to stable storage. Since sending messages across the network is considerably faster than writing to a conventional stable storage implementation, we expect write topactions to run faster in the replicated system. Let us look at the costs of running a write topaction on a system with varying degrees of replication.

In $R(1,1)$, a write topaction is just slightly more expensive than a read topaction. This difference is caused by interference between consecutive write topactions. In particular, the second phase of the commit protocol at a participant interferes with the processing of the next write topaction at the participant. Since there is no phase two at a participant where a topaction is read-only, this interference does not occur for read topactions.

In $R(1,3)$, write topactions take approximately the same time as read topactions. One would expect the interference visible in $R(1,1)$ to show up in $R(1,3)$ too. However, when the participant guardian group has three cohorts, the extra time needed to force event records to a sub-majority masks the interference between consecutive write topactions. Therefore, in $R(1,3)$, write topactions are only as expensive as read topactions.

Write topactions in $R(3,1)$ are 20 ms more expensive than write topactions in $R(1,1)$. Forcing the *committing* record at the coordinator accounts for this difference in cost. In $R(1,1)$, this force is very cheap because the coordinator has no backups. In $R(3,1)$, the force is more expensive because the coordinator has two backups and the force has to go to a sub-majority (one) of the backups. The same fact accounts for the 20 ms difference between write topactions in $R(1,3)$ and $R(3,3)$.

Top-action	Orig-inal	Replicated			
		1,1	1,3	3,1	3,3
Read (10)	241	260	290	262	291
Write (10)	340	272	302	294	322
Read (100)	2266	2472	2543	2477	2554
Write (100)	2484	2500	2590	2500	2600

Table 4.3: The time (in milliseconds) required to run a topaction consisting of multiple handler calls. Original refers to the unreplicated system. a, b refers to a replicated system with a replicas of the client and b replicas of the server. The number in the leftmost column gives the number of handler calls comprising a single topaction.

Multiple Handler Calls

Table 4.3 presents performance figures for topactions consisting of multiple handler calls. These measurements are interesting because we can expect topactions to consist of multiple handler calls. The cost of committing topactions with multiple handler calls is amortized over the different handler calls. Therefore, the overall cost per handler call should decrease as the number of handler calls per topaction is increased.

The measurements in Table 4.3 are what would be expected given the performance of handler calls in Table 4.1 and single handler call topactions in Table 4.2. As the number of handler calls per topaction increases, the time spent making handler calls begins to dominate the time spent during two phase commit. Therefore, since handler calls in the replicated system are slightly more expensive than handler calls in the unreplicated system, topactions consisting of a large number of handler calls are more expensive in the replicated system. For example, in $R(3,3)$ a topaction consisting of one hundred write handler calls takes 2600 ms, whereas the same topaction in O takes 2484 ms. This translates into a difference of 1.6 ms per handler call, which is reasonable, given the observed difference of 2 ms between handler calls in $R(3,3)$ and O.

4.7.3 Comparison with Isis

This section compares the performance of the replicated transaction system with the Isis replicated transaction facility [Bir85]. Each service in the Isis system is composed of several replicas. Rather than designating one replica as the primary for the entire service, any replica can act as the coordinator for a particular handler call. Since different handler calls can have different coordinators, the handler calls have to be synchronized with each other. Read handler calls are handled locally by acquiring a read lock and returning the result to the caller. Write handler calls are handled by first using an atomic broadcast protocol [BJ87] to acquire write locks at all the replicas and then performing the modifications.

The following table gives the costs of looking up and inserting new entries into a directory service consisting of three replicas.

Operation	Isis	VS
Lookup	100	28
Insert	158	30

These numbers measure the average cost per handler call (in milliseconds) of a transaction containing 25 requests. The first column gives the performance of an Isis system [Bir85] running on Sun 2/50 computers. The second column gives the numbers for a system with viewstamped replication running on MicroVAX II computers. Both systems use a 10 megabit/second ethernet for communication. The difference in performance arises mainly from the large costs of the atomic broadcast protocols used in Isis. A newer version of the Isis system is supposed to have significantly faster implementations of these protocols, but performance numbers for these new protocols were not available in time for inclusion in this thesis.

Chapter 5

View Change Algorithm

This chapter describes the view change algorithm that is invoked in response to node and network failures and subsequent recoveries. Section 5.1 gives an overview of the entire run-time system, including the components that implement the view change algorithm. Section 5.2 describes the strategy used to detect failures and recoveries. Section 5.3 presents the implementation of the actual view management algorithm.

The pseudo-code fragments presented in this chapter make extensive use of communication facilities and timeouts. Most of this usage should be self-explanatory. Appendix A describes some of the less obvious features used in the code fragments.

5.1 System Overview

Figure 5.1 gives an overview of the replication scheme's implementation. *Normal processing* encompasses handler call executions and two phase commits at the primary. This activity is described by a sequence of *event records* that are generated during normal processing. The event records are transmitted to the backups by the *Sender* processes. Each sender process communicates with a *Receiver* process at one of the backups. The receiver processes receive event records from the primary and store them locally in the event log. Whenever a receiver process receives an event record marking the start of a new view, it notifies the *View manager* process, which makes the cohort a backup member of the new view.

The *Prober* process runs at each cohort; it detects failures and subsequent recoveries of other cohorts in the guardian group by communicating with their prober processes. In response to such a failure or recovery, the prober process notifies the view manager process, which initiates a view change. This view change protocol is implemented by communication between the view manager processes running at different cohorts.

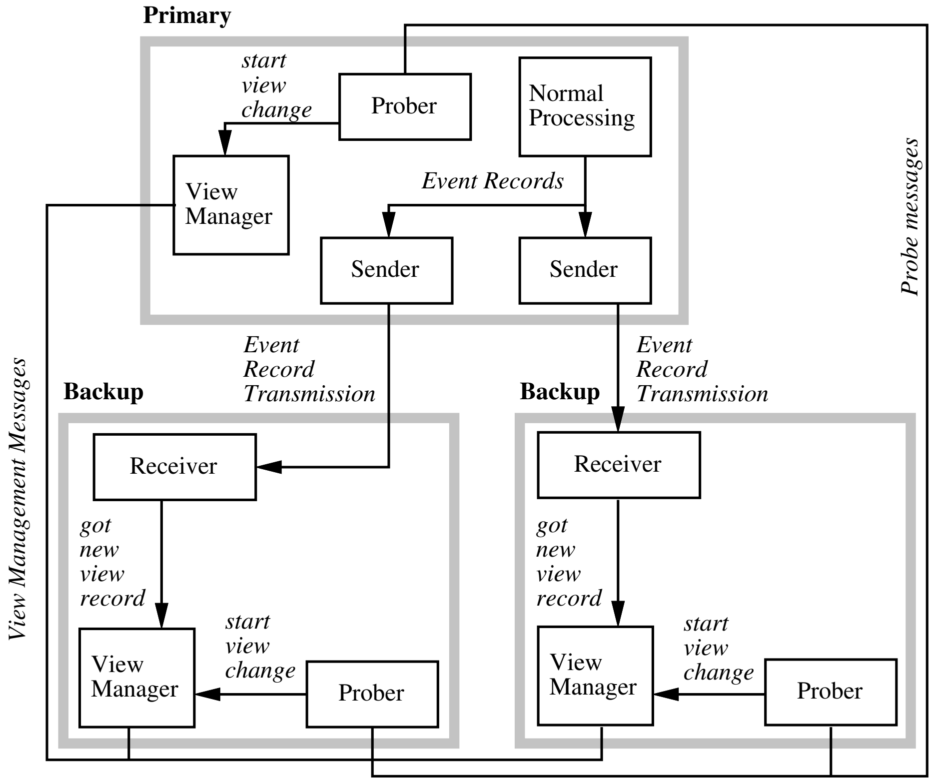

Figure 5.1: Implementation Overview.

5.2 Failure and Recovery Detection

Each cohort periodically sends *I'm alive* messages via its *Prober* process to all the other cohorts in the configuration. If cohort A does not receive any such message from cohort B in a given amount of time, then A assumes B has crashed. B may not have actually crashed; it may just have been partitioned from A by a communication link failure.

Node and communication link failures are detected by the absence of *I'm alive* messages. Node and communication link recoveries are detected by a cohort when it receives an *I'm alive* message from a cohort that was previously assumed to have crashed. The rest of this section describes the failure and recovery detection mechanism in more detail.

Each cohort A maintains some state information for failure and recovery detection (see Figure 5.2). The set *alive_cohorts* contains the guardian identifiers for the cohorts that A considers alive. *Last_msg* maps from each cohort B to the time when the most recent *I'm alive* message from B was received by A; see Appendix A for an explanation of the *map* datatype.

```
send_delay:     time_delta      % Interval to wait between sending messages
death_timeout:  time_delta      % Cohort dead if no msg in this much time
my_id:          gid             % My guardian id
other_cohorts:  set[gid]        % Guardian group configuration − { my_id }
alive_cohorts:  set[gid]        % Set of cohorts considered alive
last_msg:       map[gid,time]   % Map from cohort id to time last msg received
```

Figure 5.2: State maintained at each cohort for failure and recovery detection.

```
probe_sender = process
   while true do
      for cohort: gid in other_cohorts do
         send alive(my_id) to cohort.probe_receiver
         end
      wait until current_time() + send_delay
      end
   end probe_sender
```

Figure 5.3: Process that periodically sends I'm alive messages to all other cohorts in the configuration.

Three processes at each cohort provide failure and recovery detection. The *Prober* process in Figure 5.1 is the composition of these three processes. The *probe-sender* process (see Figure 5.3) sends *I'm alive* messages to all other cohorts every *send_delay* time units.

The *probe-receiver* process (see Figure 5.4) receives *I'm alive* messages from sender processes at other cohorts. It updates the state variables *alive_cohorts* and *last_msg* in response to these messages. In addition, when a message is received from a cohort that is considered crashed, i.e., it is not an element of the set *alive_cohorts*, a view change is initiated.

The *failure-detector* process (see Figure 5.5) waits until it locates a cohort in the *alive_cohorts* set from which an *I'm alive* message has not been received for the last *death_timeout* time units. It removes this cohort from the *alive_cohorts* set and initiates a view change. In other words, if a message is not received from a cohort for *death_timeout* time units, the cohort is assumed to have crashed and a view change is initiated. *Death_timeout* should be several times as large as *send_delay* so that lost *I'm alive* messages do not cause unnecessary view changes.

There are conflicting requirements that influence the choice of *send_delay* and *death_timeout*. On the one hand, these values should be large so that the failure and recovery detection mechanism is activated infrequently and does not interfere with transaction processing. On the other hand, these values should be small so that failures are detected quickly. The correct choice

```
probe_receiver = process
    while true do
        receive
          alive(cohort: gid):
            % Update the time last message was received from cohort
            last_msg[cohort] := current_time()

            % I'm alive message from cohort
            if cohort ∉ alive_cohorts then
                % Dead cohort is now alive
                alive_cohorts := alive_cohorts ∪ { cohort }
                send start_view_change to my_id.view_manager
                end
      end
    end probe_receiver
```

Figure 5.4: Process that receives I'm alive messages and detects cohort recoveries.

```
failure_detector = process
    while true do
        % Check if there is a cohort from which a message hasn't been received recently
        for cohort: gid in alive_cohorts do
            if (current_time() − last_msg[cohort]) ≥ death_timeout then
                alive_cohorts := alive_cohorts − {cohort}
                send start_view_change to my_id.view_manager
                end
            end
          % Wait until time to check again
          time_of_next_death: time := min {last_msg[c] | c ∈ alive_cohorts} + death_timeout
          wait until time_of_next_death
          end
    end failure_detector
```

Figure 5.5: Process that detects cohort failures.

of values for these two system parameters is an engineering decision that should be made after careful consideration. In my implementation, *send_delay* is five seconds and *death_timeout* is five times the value of *send_delay*. *Death_timeout* is made significantly larger than *send_delay* because some *I'm alive* messages may be lost because we use an unreliable message transport mechanism.

```
% State kept for view management
type status = oneof [
    active,          % Cohort is member of current view
    view_manager,    % Cohort is view change manager
    inactive         % Cohort is none of the above
    ]

state:       status   % Current cohort state
cur_view:    view     % Most recent view entered by this cohort
max_viewid:  viewid   % Largest viewid seen at this cohort
crashed:     bool     % True if recovering from a crash
```

Figure 5.6: State information used by the view change algorithm.

5.3 The Algorithm

The view change algorithm is invoked by the failure and recovery detection mechanism as described in the previous section. The cohort where the algorithm is initiated becomes the view change manager. It sends out invitations to all the other cohorts in the group; the other cohorts reply to the invitations with acceptance messages. When the manager has received acceptances from enough cohorts, it forms a new view. The manager then sends a message to the new view's primary to inform it of the new view. A single failure or recovery may be detected by several different cohorts, each of which may initiate a view change in response; the protocol correctly handles multiple concurrent view changes.

5.3.1 State Information

Each cohort maintains state information that is needed by the view change algorithm (see Figure 5.6). The cohort can be in one of three states. A cohort is *active* when it is either the primary or a backup in some view. A cohort enters the *view_manager* state when it detects a failure or recovery and initiates a view change algorithm. A cohort is *inactive* when it is neither *active* nor a *view_manager*. *Cur_view* records the most recent view of which the cohort was a member. *Cur_view.vid* is maintained on stable storage; every time the cohort enters a new view, the modification to *cur_view.vid* is recorded on stable storage. *Max_viewid* is the largest viewid ever seen by the cohort; it is also maintained on stable storage. *Crashed* is a boolean variable that is true for any cohort that has not entered a view since it last crashed. Therefore, if *crashed* is true at cohort A, then A's event log has been lost in the crash and has not been updated since. *Crashed* and *cur_view.vid* are used during view formation to decide whether or not a new view can be formed without losing committed information.

A view management process runs at each cohort; it manages this state information, and initiates and responds to view change messages (see Figure 5.7).

5.3.2 The Active State

An *active* view management process responds to two kinds of messages. If a *start_view_change* message is received from the failure and recovery detection mechanism, the view management process enters the *view_manager* state. An *active* process also responds to invitations to join new views by calling the procedure *accept_invite*, which tries to accept the invitation; see Figure 5.8. The invitation message contains the viewid of the new view being formed. If this viewid is not greater than *max_viewid*, the largest viewid seen at this cohort, the invitation is ignored and the cohort remains in the *active* state. Otherwise, *max_viewid* is updated and an acceptance message is sent back to the view manager that sent the invitation. After sending the acceptance message, the process enters the *inactive* state.

Several pieces of information are passed in this acceptance message. First, the boolean variable *crashed* is part of the message, indicating whether or not the cohort responding to the invitation has recovered from its last crash. Second, the cohort sends the most recent view it was a part of. For cohorts that have not recovered from their last crash, only the viewid component of this view is meaningful, as it is the only component stored on stable storage. Third, the cohort sends the viewstamp of the most recent event in its event log. For cohorts that have not recovered from their last crash, this viewstamp is the viewstamp for a special event that is present at the beginning of each cohort's event log. The viewstamps sent in the acceptance messages are used by a newly selected primary to update the *extent*[*b*] variables maintained by the sender processes; see Figure 4.4. The acceptance message also contains the identity of the cohort that accepted the invitation, and the viewid of the view being formed. After sending the acceptance message, the view management process enters the *inactive* state.

5.3.3 The Inactive State

An *inactive* view management process accepts four types of messages. If it receives an invitation to join a new view, it calls the procedure *accept_invite* to respond to the invitation; see Figure 5.8. If it is notified of a failure or recovery by the failure and recovery detection mechanism, it enters the *view_manager* state and initiates a view change.

If the process receives a message from the manager of a successful view change that indicates that it should become the primary in a new view, it calls the procedure *turn_into_primary* to join

```
view_manager = process
  while true do
    tagcase state
      tag active:
          receive
           start_view_change:
             state := view_manager
           invite(new_viewid: viewid, manager: gid):
             if accept_invite(new_viewid, manager) then
                 state := inactive
                 end
           others:
             % Ignore other kinds of messages
      tag inactive:
          % Remain inactive for next view_manager_repeat time units.
          % Then try a view change
          next_attempt: time := current_time() + view_manager_repeat
          while state = inactive do
            receive before next_attempt
              invite(new_viewid: viewid, manager: gid):
                if accept_invite(new_viewid, manager) then
                    % Accepted invitation from a view manager. Delay view
                    % mgmt. for another view_manager_repeat time units
                    next_attempt := current_time() + view_manager_repeat
                    end
              become_primary(nv: view, new_extents: map[gid, viewstamp]):
                state := active
                turn_into_primary(nv, new_extents)
              start_view_change:
                state := view_manager
              new_view_event_record(nv: view, history: set[viewstamp]):
                state := active
                turn_into_backup(nv, history)
              timeout:
                % Have timed out. Try another view change
                state := view_manager
              others:
                % Ignore other messages
            end
      tag view_manager:
          do_view_change()
      end
    end
  end view_manager
```

Figure 5.7: The view management process run at every cohort.

```
% Handle invitation message. Return true iff invitation accepted.
accept_invite = proc (new_viewid: viewid, manager: gid) returns (bool)
    if new_viewid ≤ max_viewid   then
        return false
        else
        max_viewid := new_viewid
        write_to_stable_storage(max_viewid)
        send  accept(crashed, cur_view, event_log$max_vs(), new_viewid, my_id)
              to manager.view_manager
          return true
          end
    end accept_invite
```

Figure 5.8: Accepting an invitation to join a new view.

```
turn_into_primary= proc (new_view: view, backup_extents: map[gid, viewstamp])
    cur_view := new_view
    write_to_stable_storage(cur_view.vid)
    backups := cur_view.backups
    execute_log()
    vs: viewstamp := event_log$append(new_view_record(cur_view, vs_history))
    for backup: gid in backups do
        extent[backup] := backup_extents[backup]
        end
    % Send new view record to all and then become active
    force_to_all({vs})
    end turn_into_primary

turn_into_backup = proc (new_view: view, primary_history: set[viewstamp])
    event_log$trim_to(primary_history)
    vs_history := primary_history
    cur_view := new_view
    write_to_stable_storage(cur_view.vid)
    end turn_into_backup
```

Figure 5.9: Entering a new view.

the new view as a primary; see Figure 5.9. *Turn_into_primary* updates the *extent*[*b*] variables maintained by the sending processes (see Figure 4.4) and generates a new view event record containing the new view and the primary's viewstamp history.

If the new primary was previously a backup, some of the event records in its event log may not have been applied to its state. Therefore, the new primary invokes the procedure *execute_log* to update its state to correspond to the set of events in the event log. The procedure *force_to_all* is then called to send the contents of the entire event log to all the backups. *Force_to_all* is the same *force* (see Figure 4.5), except that *force_to_all* forces the events to all of the backups instead of just a sub-majority. Just like *force*, *force_to_all* sends a portion of the event log to a backup only if the backup does not already have that portion.

If a *receiving* process (see Figure 5.1) receives a new view event record from the sender process at a newly selected primary, it notifies the *inactive* view management process, which in turn calls the procedure *turn_into_backup* to join the new view as a backup; see Figure 5.9. The viewstamp history passed in the new view event record is used to update the cohort's own viewstamp history, and to trim portions of the cohort's event log that are not present at the primary. Both *turn_into_backup* and *turn_into_primary* update the variable *cur_view* and write *cur_view.vid* to stable storage to record the cohort's membership in the new view.

If the *inactive* view management process does not receive any of these messages within a given interval of time, it times out and initiates a new view change by entering the *view_manager* state.

5.3.4 The View Manager State

The behavior of a view management process in the *view_manager* state is shown in Figure 5.10. The manager starts out by creating a new viewid *new_vid* that is greater than all other viewids seen by the manager. All viewids have two fields, a counter and a tag. The new viewid is generated by incrementing *max_viewid.counter* and tagging it with the manager's guardian identifier. The viewid is ordered lexicographically by these two fields; i.e., $\langle \mathrm{c1.g1} \rangle < \langle \mathrm{c2.g2} \rangle$ if $c1 < c2$ or $(c1 = c2$ and $g1 < g2)$. The tag field guarantees that viewids generated at different cohorts will be different. The counter field guarantees that the new viewid will be greater than *max_viewid* and therefore greater than all other viewids seen by the manager. *Max_viewid* is set to *new_vid* and then written to stable storage. The manager then sends invitations to all other cohorts in the group and waits for their replies. The replies are stored in *responses*, which maps cohort guardian identifiers to the replies. The manager stops waiting for replies when a reply has been received from each cohort. A timeout, *accept_wait*, is built-in so that if a cohort is unreachable due to a guardian or communication link failure, the manager does not wait too long for it to respond. In the current implementation, *accept_wait* is one second. In addition to timing out, the manager stops waiting if it receives an invitation to join a view with a higher viewid than *new_vid*. An acceptance is sent back in response to this invitation and the view manager process enters the *inactive* state.

After the responses have been collected, a new primary is selected from the set of responding cohorts. The new primary should have more information than all other cohorts that responded

```
% Information stored about acceptances in response to invitations
type response = record [ last_view: view, max_vs: viewstamp, crashed: bool ]

do_view_change = proc ()
  new_vid: viewid := make_viewid{counter: max_viewid.counter + 1, tag: my_id}
  max_viewid := new_vid
  write_to_stable_storage(max_viewid)
  for cohort: gid in configuration − {my_gid} do
     send invite(new_vid, my_gid) to cohort.view_manager
     end
  responses: map[gid,response]
  responses[my_gid] := make_response(cur_view, event_log$max_vs(), crashed)

  % Wait for acceptances that can be received in time acceptwait
  finish_waiting: time := current_time() + accept_wait
  waiting: bool := true
  while waiting do
    receive before finish_waiting
     start_view_change: % Restart view management
      return
     accept(crash: bool, last_view: view, max_vs: viewstamp, vid: viewid, id: gid):
      if vid = new_vid then % Responding to current view change
         responses[id] := make_response(last_view, max_vs, crash)
         if |responses| = |configuration| then
            % Got responses from everybody − stop waiting
            waiting := false
            end
         end
     invite(new_viewid: viewid, manager: gid):
      if accept_invite(new_viewid, manager) then
         state := inactive
         return
         end
     timeout: % Stop waiting for acceptances
      waiting := false
     others: % Ignore other messages
    end

  primary: gid := find_primary(responses)
  if can_form_view(primary, responses) then
    % Make view
    v: view := make_view(primary, vid, {c | c ∈ responses} − {primary})
    extents: map[gid, viewstamp] := {c → vs | c ∈ v.backups and vs = responses[c].max_vs}
    send become_primary(v, extents) to primary.view_manager

     state := inactive
     end
  end do_view_change
```

Figure 5.10: The view management process at a view change manager.

```
can_form_view = proc (primary: gid, responses: map[gid, response]) returns (bool)
   last_view: view := responses[primary].last_view
 1: if not is_majority({c | c ∈ responses}) then
       return false
       end
 2: if not is_majority({c | responses[c].last_view.vid ≤ last_view.vid}) then
       return false
       end
   not_crashed: set[gid] := {c | not responses[c].crashed}
   old_crashed: set[gid] := {c | responses[c].crashed and responses[c].max_vs.vid < last_view.vid}
 3: if last_view.primary ∈ not_crashed then
       return true
       end
 4: if is_majority(not_crashed ∪ old_crashed) then
       return true
       end
   return false
   end can_form_view
```

Figure 5.11: Deciding whether a new view can be formed.

---

to the invitation; i.e., the *max_vs* field in its response message should not be smaller than the *max_vs* field of any other response to the invitation. This procedure will automatically choose a non-crashed cohort over a crashed cohort because crashed cohorts return the smallest viewstamp possible.

After the new primary has been selected, the collected responses are checked to see if a new view can be formed; see Figure 5.11. The conditions checked for allowing a view formation are described in Section 5.3.5. If these conditions are met, the new primary is notified of the new view. If for some reason the new view cannot be formed, the view management process becomes *inactive*. *Inactive* processes eventually timeout and initiate a new view change protocol.

5.3.5 Forming a New View

There are two properties that must hold for all views. First, the view should contain at least a majority of the cohorts in the configuration. Second, all previously committed information should be present at the primary of the new view. The view formation conditions in Figure 5.11 ensure that the new view has the two required properties. These conditions are a relaxation of the ones described in [Oki88] and were developed during a discussion with Robert E. Gruber. The algorithm in Figure 5.11 checks that the new view has the first property in the obvious way.

The rest of the algorithm ensures that all committed information is present at the new primary. The variable *last_view* is used in the rest of this discussion to refer to the last view of which the new primary was a member.

We can divide committed information into three categories — *committed_in_last, committed_before_last*, and *committed_after_last*. *Committed_in_last* refers to information committed during *last_view*, *committed_before_last* refers to information committed before *last_view*, and *committed_after_last* refers to information committed after *last_view*.

Since the new primary has not seen any views after *last_view*, it does not have any *committed_after_last* information. Therefore, if there is a possibility that *committed_after_last* information exists, the view formation cannot be allowed to succeed. No information can have been committed after *last_view* if a majority of the cohorts have not seen a view after *last_view*. Therefore, test number 2 in the algorithm allows the view formation to proceed only if a majority of the cohorts have not seen a view after *last_view*.

Consider *committed_before_last* information. If we assume that the view formation that led to the formation of *last_view* was correct, then *last_view.primary* had all the *committed_before_last* information at the start of *last_view*. As part of the view change protocol, this committed information was made available to all of the members of *last_view*, including the new primary. Therefore, the new primary is guaranteed to have all the *committed_before_last* information.

Consider the *committed_in_last* information. If *last_view.primary* is a non-crashed member of the new view, then it has all the *committed_in_last* information. Since the new primary is supposed to have more information than all the other cohorts in the new view, the new primary also has all the *committed_in_last* information. Therefore, test number 3 allows the view formation to succeed if *last_view.primary* is a non-crashed member of the new view.

If *last_view.primary* has crashed since the formation of *last_view*, some *committed_in_last* information may have been lost because all the members of *last_view* that had this committed information may have crashed either during or after *last_view*. However, if the non-crashed cohorts and the cohorts that crashed before *last_view* form a majority, then we know that the cohorts that crashed during or after *last_view* do not form a majority. Therefore, since all committed information is written to a majority of the cohorts, at least one of the cohorts with all the *committed_in_last* information is present in the new view. Since the new primary is supposed to have more information than all the other cohorts in the new view, the new primary also has all the *committed_in_last* information. Therefore, test number 4 allows the view formation to succeed if the non-crashed cohorts and the cohorts that crashed before *last_view* form a majority.

5.4 Performance

This section describes the performance of the view change algorithm. There are several factors that contribute to the overall cost of a view change.

1. The view change manager writes the new *max_viewid* to stable storage and sends invitations to all other cohorts.

2. The new *max_viewid* is written to stable storage by the cohorts that decide to accept the invitations.

3. The manager waits at most *accept_wait* time units for acceptances from the other cohorts.

4. The manager send the new view information to the new primary.

5. The new primary updates its state by executing the portion of the event log that it has received since the last time it was a primary.

6. The new primary writes *cur_view.vid* to stable storage.

7. The new primary forces the contents of the event log to all the backups.

The main costs here are the writes to stable storage, the execution of the event log and the force of the log to all the backups. Since the cost of writes to stable storage is fairly insignificant compared to *accept_wait*, stable storage writes can be ignored when discussing the cost of view changes.

5.4.1 Executing the Event Log

The cost of executing the event log depends on the size of the portion that needs to be executed. If the new primary was also the primary in the previous view then the entire event log will already have been executed and this cost will be zero. However, in certain situations the new primary will need to execute the entire event log. Since it takes approximately 18 seconds in the current implementation to execute one megabyte of the log, view changes become very costly as the log gets large. One possible way to avoid this cost is to execute event records as they are inserted into the event log at a backup, instead of waiting for a view change before executing the entire event log in one shot. The disadvantage of this method is that it increases the amount of computation that needs to be performed by a backup in response to the receipt of an event record from the primary. This method is not part of the current implementation.

5.4.2 Forcing the Log to the Backups

The cost of forcing the event log to the backups depends on how out of date the backups' event logs are. If all the logs at the backups are fairly up-to-date with respect to the primary's event log, then the force does not take very long. On the other hand, a backup that has just recovered from a crash does not have anything in its log and the primary's entire event log has to be sent to the backup. In a view with two backups, sending a megabyte of information to both backups takes approximately 8 seconds. Therefore, it is quite costly to force the event log to backups that are significantly out of date. One solution to this problem is to store the event log on non-volatile storage. Event logs on non-volatile storage will survive normal node crashes and newly recovered cohorts will not be completely out of date with respect to the primary's event log. If the event log on non-volatile storage is destroyed by a media failure, then the system can fall back to the original method. It should be relatively straightforward to add this optimization to the current implementation.

5.5 Optimizations

The cost of the protocols presented in this chapter can be reduced in several ways. First, the failure detection scheme described in Section 5.2 involves sending a message from each cohort to every other cohort in the configuration. Therefore, in a configuration with n cohorts, $n(n-1)$ failure detection messages are sent every *send_delay* time units. Section 5.5.1 presents several schemes to reduce this cost. Second, a single failure or recovery can be detected by many cohorts. Therefore, multiple view change protocols can be started in response to a single failure or recovery. Section 5.5.2 describes several schemes that try to minimize the number of view change protocols started in response to a single failure or recovery. Third, view changes run faster if we keep the log small. This is described in Section 5.5.3.

Since view changes are assumed to be rare — they only occur as a result of node or communication failures — the cost of view changes is not as important as the impact of the replication scheme on normal processing. Therefore, the implementation has been tuned for efficient normal processing. The optimizations described in this section have not been implemented.

5.5.1 Reducing the Cost of Failure Detection

There are several mechanisms that can be used to reduce the number of messages sent for failure detection. Two are described below.

1. *I'm alive* messages can be piggy-backed on other messages. Since a primary periodically sends event records to the backups and receives acknowledgments, (see Figure 4.4), this technique can significantly reduce the number of explicit *I'm alive* messages sent between the primary and the backups. Since the *I'm alive* messages contain very little information, the piggy-backing imposes very little extra cost on normal message transmission.

2. Backups try to detect just their primary's failure. Primaries and *inactive* cohorts try to detect failures and recoveries of all other cohorts. If all the cohorts are *active* and part of the same view, this mechanism will cut down the number of *I'm alive* messages sent each *send_delay* time units to $2(n-1)$ because the primary will send $n-1$ messages (one to each backup), and the $n-1$ backups will each send a message to the primary.

5.5.2 Preventing Concurrent View Managers

The view change algorithm performs correctly in the presence of multiple concurrent view change managers [Oki88]. However, it is desirable to minimize the number of concurrent view changes initiated in response to a failure or recovery detection, as concurrent view changes can interfere with each other and delay the formation of a new view. One way to achieve this is to assign different values of *death_timeout* to all the cohorts in the guardian group. For example, consider the configuration $\{g1, g2, g3, \ldots\}$. Let $g1$'s *death_timeout* value be smaller than all the other cohorts' values. Suppose some cohort fails at time t. If $g1$ is alive, this failure will probably be detected by $g1$ before it is detected by another cohort. Therefore $g1$ will initiate a view change and notify the other cohorts of the failure before any of the other cohorts have a chance to detect the failure first-hand and initiate a view change of their own.

The second scheme described in Section 5.5.1 also reduces the number of concurrent view changes initiated in response to certain failures. If the failed cohort is a backup, the failure will be detected by just the primary and only one view change will be initiated. However, failures of *inactive* cohorts and the primary might still result in the initiation of multiple view changes.

5.5.3 Log Compaction

Since the costs of executing the event log and updating the event logs at the backups are proportional to the log size, view changes can be made faster by keeping the log small. Standard log compaction techniques can be used to achieve this. A log compaction technique involves the periodic application of transformations that reduce the size of the log. For example, one useful transformation discards the *completed call* event records for subactions of aborted topactions.

Another transformation discards the effects of a handler call that have been superseded by later handler calls. These transformations can keep the size of the log very close to the amount of state being maintained by the guardian group. Since the log can be used to generate the guardian group state, and therefore the size of the state is a lower bound on the log size, we cannot hope to do better than this.

Chapter 6

Conclusions

This thesis presents an implementation of a replication scheme for constructing highly available services. Each service is composed of several *cohorts* (or replicas). One of these cohorts is designated the *primary* and the others are the *backups*. The primary handles all requests made to the service and propagates information about these requests to the backups. If the primary crashes, a backup takes over.

The replication scheme can be divided into two parts, *normal processing* and *view changes*. During normal processing, the primary handles requests from clients. Servicing a request may result in modifications being made to the service state. These modifications are recorded in *event records*, which are propagated to the backups. Cohort and communication link failures are handled by invoking a view change algorithm that results in the selection of a new primary and a new active set of backups for this primary. Normal processing continues once the view change is finished. The service remains available as long as a majority of the cohorts that constitute the service are up and able to communicate with each other.

The thesis demonstrates that viewstamped replication is a feasible low-cost method for constructing highly available nested transaction based services. Chapter 4 contains a detailed discussion of the overhead of replication.

The implementation focuses on reducing the overhead imposed by the replication scheme during normal processing. Normal processing consists of handler calls and topaction commits. Handler call processing in the replicated system is as efficient as in the unreplicated system (see Table 4.1). The slight overhead in the replicated system arises mainly from the creation and transmission of event records. The other factors that contribute to the increased cost of replicated handler calls are given below.

- Before sending the handler call message, the client locates the server's current primary by using the mechanism described in Section 4.4.
- A viewstamp set is sent from the server primary to the client in the handler call reply message.
- The client has to maintain a viewstamp set for each topaction that is currently active; this set needs to be updated after every handler call.

The replication scheme must ensure that information about the handler calls made for a committing topaction is known at a majority of the cohorts. This information is transmitted to the backups by sending the appropriate event records and waiting for acknowledgments. The scheme used for the reliable transmission of these event records guarantees that usually the acknowledgments will be received after a message round trip delay. In the unreplicated system, since there are no backups, the information about handler calls made for the committing topaction is stored reliably by writing it to stable storage. Since writes to stable storage are generally more expensive than message round trip delays on a local area network, topaction commits are cheaper in the replicated system.

The failure detection system and the view change protocol have been implemented and thoroughly tested. However, their implementation has not been fully optimized. View changes can be slow after the service has been up for some time; see Section 5.4 for an explanation. Section 5.5.3 presents an optimization that can significantly speed up view changes.

In a guardian group with n replicas, the failure detection system can send as many as $n(n-1)$ messages during each failure detection interval. Section 5.5.1 presents two methods that can reduce the number of these messages. The failure detection system can also start multiple view changes at different cohorts in response to a single failure or recovery. Section 5.5.2 describes two methods that alleviate this problem by decreasing the number of view changes started in response to a single failure or recovery.

6.1 Contributions

This thesis demonstrates that viewstamped replication provides a low-cost method for constructing highly available nested transaction based services. This claim is supported by the performance measurements and the comparison with an unreplicated transaction system given in Section 4.7. The contributions of this work are:

- Viewstamped replication is integrated into the Argus run-time system to automatically provide high availability for transaction based services. The programmer designs a service without worrying about availability; the run-time system automatically replicates this service to make it highly available.

- The implementation of the replication scheme imposes little overhead on service operation. In particular, during normal processing the efficiency of the replicated service is close to that of the unreplicated service.

- Several performance measurements of the replication scheme are given (see Section 4.7). These measurements are analyzed and compared with corresponding measurements of the performance of an unreplicated system.

- The conditions required for forming a new view as given in [Oki88] were more stringent than required. The thesis relaxes these conditions (see Figure 5.11 and Section 5.3.5) and allows view formations in cases where the original conditions would not have allowed a view formation.

- A clean break-down of the replication scheme into different components is given (see Figure 5.1).

- The original replication scheme would transfer the entire service state when sending information from a new primary to the backups [Oki88]. The thesis describes a mechanism that solves this problem by transferring only the differences between the replica states. This mechanism uses the *event log* and the *sender* and *receiver* processes.

- A location service for identifying the current primary of a replicated service is described and implemented (Section 4.4).

6.2 Extensions

This section presents some extensions to the viewstamped replication scheme and suggests some directions for future work.

Improving data resiliency. In the unreplicated system, all committed information is stored on stable storage and is therefore highly resilient to failures. In the replicated system, there are copies of all committed information at a majority of the replicas. If these replicas crash simultaneously, the committed information will be lost. This is a short-coming of the system, and a mechanism for making the service state resilient to simultaneous failures is needed.

We can make the replicated service resilient to failures by attaching disks to the replicas, and waiting for the necessary information to reach at least two disks before allowing a topaction to commit. Since most stable storage implementations provide resiliency by writing information to two disks, this implementation is as resilient to failures as the unreplicated transaction system that uses stable storage.

The increased resiliency to failures does not come without a cost. Topaction commits have to be delayed while the necessary information is written to disks. However, a slight modification to the scheme can remove this extra cost. If we attach uninterruptible power supplies to all the replicas, we do not have to delay the topaction commit until the necessary information reaches the disks. The information can be written to the disks asynchronously. The uninterruptible power supply guarantees that if some information reaches a replica, it will make it out to disk before the replica crashes due to a power failure. Therefore, a topaction commit can be allowed to complete as soon as the necessary information has reached at least two replicas with uninterruptible power supplies and attached disks.

The cost of this mechanism lies in the uninterruptible power supplies and the disks required for resiliency. The uninterruptible power supplies are optional, because we can get by with synchronous writes to disk, as long as we are prepared to pay the extra cost of topaction commits. However, it is not possible to use this mechanism on disk-less workstations. On disk-less workstations, the service should probably use a network based stable storage service to provide high resiliency to failures.

Using stable storage. [DST87, Coh89] describe highly available network-based stable storage services. These services achieve high availability by replicating the stable storage servers and using a voting scheme [Gif79] to access the replicas.

Such stable storage services can be used to provide highly available and reliable general purpose services. Consider an unreplicated Argus guardian that uses a stable storage service for resiliency. If this guardian fails, the service it provides is not available until the guardian recovers and re-creates its state from stable storage. However, after the guardian fails, a new guardian can be created at another location, and the old guardian's state can be re-created at the new guardian from stable storage. The new guardian now handles all service requests on behalf of the old guardian. Therefore, the highly available stable storage service can be used to create highly available general purpose services in a straightforward manner.

Details of how a new guardian takes over from an old one need to be worked out. In particular, some form of a view change algorithm is required to replace the original guardian after it crashes. In addition, a location service [Hwa87] is needed to route requests to the new guardian.

It would also be interesting to see if the voting scheme used for implementing the replicated stable storage services should be replaced by a primary copy replication scheme. The relative merits of the two schemes in this particular instance are not known.

Replicas without data. Both voting and viewstamped replication schemes require a minimum of three replicas to provide higher availability than an unreplicated system. However, two copies of data are enough to make it resilient to many failures (most stable storage implementations use two disks to store each piece of data).

[P$\hat{8}$6] proposes an extension to voting schemes where some replicas do not store the service state; they just provide votes for operations. The replicas without service state are called *witnesses*; the replicas with service state are *copies*. Under very general assumptions, the reliability of a replicated service consisting of n copies and m witnesses is shown to be the same as the reliability of a service consisting of $n + m$ copies. Under most circumstances, the availability of the system is also not significantly reduced when some copies are replaced with witnesses.

[MHS89] and [GL89] adapt the witness scheme for primary copy replication. In primary copy replication, witnesses participate only in primary elections (view changes); they do not store any service state. Witnesses are an attractive extension to viewstamped replication because they significantly decrease the amount of resources used by a service without significantly affecting either its reliability or availability.

Using time to avoid communication. Section 4.6.4 describes why the optimization made in the unreplicated system for the first phase of two phase commit for read-only transactions cannot be applied in the replicated system. This optimization could not be made because the primary could not be sure whether or not a new primary had been selected without its knowledge. There is a simple solution to this problem [MHS89, Lis89]. Each backup periodically makes a promise to the primary saying that it will not enter a new view for the next n seconds. This information allows the primary to place a lower bound on the earliest time at which a new view can be formed. If the current time is smaller than this lower bound, then the primary knows that no new view has been formed yet. Therefore, it does not need to force the read topaction's event records to a majority before allowing the topaction to commit. This technique can make read-only transaction commits as efficient as in the unreplicated system. However, it requires that the clocks at the different cohorts either be synchronized, or that they run at nearly identical rates.

Appendix A

Pseudo-code Syntax

Code fragments presented in this thesis make extensive use of communication facilities and several data-types that are not built-in to most languages. This appendix gives an informal description of some of the non-standard features used in the pseudo-code presented in this thesis.

A.1 Data Types

This thesis uses some data-types that deserve a detailed explanation.

Time related data types. Two time related data-types, *time* and *time_delta*, are used in this thesis.

- Values of type *time* represent absolute time in an unspecified time unit.
- The function *current_time()* returns the current time.
- Values of type *time_delta* represent the difference between two time values.
  - Subtracting one *time* value from another gives a *time_delta* value.
  - Subtracting or adding a *time_delta* value to a *time* value gives a *time* value.
- Addition and subtraction of two *time_delta* values results in another *time_delta* value.
- Multiplication and division of a *time_delta* value by an integer or a real number results in a *time_delta* value.
- Values of type *time* form a totally ordered set, as do values of type *time_delta*.

- Special values of type *time_delta* corresponding to *seconds* and other standard units of time are available.

Maps. Values of type *map* provide a mapping from a subset of values of one type to values of another type. For example, a value of type *map*[*string,int*] maps from strings to integers.

- Maps are usually created empty; i.e., they do not provide a mapping for any value.

- Initialized maps of type *map*[*a,b*] can be created by providing a set of bindings from values of type *a* to values of type *b*. For example,

$$\text{m: map[string,int]} \leftarrow \{(\text{“a”} \rightarrow 1), (\text{“b”} \rightarrow 2), (\text{“c”} \rightarrow 3)\}$$

- The $\in$ operation can be used to check whether a map provides a mapping for a certain value. With the example shown above, “a” $\in$ m will be true; “x” $\in$ m will be false.

- New mappings can be inserted and old ones modified with an array assignment syntax. For example,

$$\text{m[“f”]} \leftarrow 19$$

$$\text{m[“a”]} \leftarrow 7$$

- Mappings can be looked up with array lookup syntax; with map *m* initialized as shown above, i: int $\leftarrow$ m[“b”] assigns the value 2 to the variable *i*.

A.2 Waiting

Two features allow a process to wait for either a given condition, or until some time.

wait for ***condition*****.** Delay the process until the boolean-valued expression *condition* becomes true.

wait until ***time-value*****.** Delay the process until the time specified by the expression *time-value*. For example, “**wait until** current_time() + second” delays the process for one second.

A.3 Communication

Processes at a given cohort can send messages to specific processes at other cohorts. Processes arrange to receive messages by waiting for them.

send $\langle$*message*$\rangle$ **to** $\langle$*gid*$\rangle$. $\langle$*pid*$\rangle$**.** Send $\langle$*message*$\rangle$ to the process named $\langle$*pid*$\rangle$ at cohort $\langle$*gid*$\rangle$. The message consists of a name and some values. For example,

employee_number("John F. Doe", 12345)

is a legal message. The message transmission is unreliable; the message may be delayed, duplicated or never delivered.

receive $\langle$*message-handlers*$\rangle$**.** Wait until a message is available and then invoke the appropriate handler from $\langle$*message-handlers*$\rangle$ based on the name of the message received. With the code shown here, a receipt of the message shown above will insert the mapping ("John F. Doe" $\rightarrow$ 12345) in *database*.

```
receive
  address(name: string, addr: string):
    ...
  employee_number(name: string, number: int):
    database[name] ← number
  others:
    ...
  end
```

This code provides message handlers for employee_number and address messages. Other messages are handled by the **others** branch.

receive before $\langle$*time*$\rangle$ $\langle$*message-handlers*$\rangle$**.** Waits until time $\langle$*time*$\rangle$ for a message. If a message is received, the appropriate message handler in $\langle$*message-handlers*$\rangle$ handles it. A special message handler named **timeout** is invoked when no message is received in the allotted time.

Bibliography

[AD76] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of Second National Conference on Software Engineering*, pages 562–570. IEEE, 1976.

[Bar78] Joel F. Bartlett. A 'nonstop' operating system. In *Eleventh Hawaii International Conference on System Sciences*, pages 103–117, January 1978.

[Bar81] 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.

[BBG83] 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.

[Bir85] Kenneth P. Birman. Replication and fault tolerance in the isis system. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 79–86. ACM SIGOPS, December 1985.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failure. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[Coh89] Jeffery I. Cohen. Distributed atomic stable storage. Master's thesis, MIT, January 1989.

[Coo85] 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.

[DST87] Dean S. Daniels, Alfred Z. Spector, and Dean S. Thompson. Distributed logging for transaction processing. In *In ACM Special Intrerest Group on Management of Data 1987 Annual Conference*, pages 82–96. ACM SIGMOD, May 1987.

[ESC85] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings of the Fourth Symposium on Principles of Data Base Systems*, pages 215–229. ACM, 1985.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, CA, December 1979. ACM SIGOPS.

[GL89] Robert Gruber and Barbara Liskov. Witnesses. Mercury Design Note 36, Programming Methodology Group, MIT Laboratory for Computer Science, Cambridge MA, November 1989.

[GLPT76] J. N. Gray, R. A. Lorie, G. F. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In G.M. Nijssen, editor, *Modeling in Data Base Management Systems*. North Holland, 1976.

[Gra78] J. Gray. Notes on database operating systems. In *Operating Systems – An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[Her86] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[Hwa87] D. J. Hwang. Constructing a highly-available location service for a distributed environment. Technical Report MIT/LCS/TR-410, M.I.T. Laboratory for Computer Science, Cambridge, MA, November 1987. Master's thesis.

[Lam81] B. Lampson. Atomic transactions. In *Distributed Systems: Architecture and Implementation*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer-Verlag, Berlin, 1981.

[LCJS87] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of argus. In *Proc. of the 11th Symposium on Operating Systems Principles*, Austin, Tx, November 1987. ACM.

[Lis88] B. Liskov. Distributed programming in argus. *Comm. of the ACM*, 31(3):300–312, March 1988.

[Lis89] Barbara Liskov. Using time to avoid communication. Mercury Design Note 37, Programming Methodology Group, MIT Laboratory for Computer Science, Cambridge MA, December 1989.

[MHS89] T. Mann, A. Hisgen, and G. Swart. An algorithm for data replication. Report 46, DEC Systems Research Center, Palo Alto, CA, June 1989.

[Mos81] J.E.B. Moss. Nested transactions: an approach to reliable distributed computing. Technical Report MIT/LCS/TR-260, M.I.T. Laboratory for Computer Science, Cambridge, MA, 1981. PhD thesis.

[Nel81] B. Nelson. Remote procedure call. Technical Report CMU-CS-81-119, Carnegie Mellon University, Pittsburgh, Pa., 1981.

[Oki88] B. M. Oki. Viewstamped replication for highly available distributed systems. Technical Report MIT/LCS/TR-423, M.I.T. Laboratory for Computer Science, Cambridge, MA, May 1988. PhD thesis.

[P$\hat{8}$6] Jehan-François Pâris. Voting with witnesses: A consistency scheme for replicated files. In *Proceedings of the 6th International Conference on Distributed Computer Systems*, pages 606–612. IEEE, 1986.

[Sch83] F. B. Schneider. Fail-stop processors. In *Digest of Papers from Spring Compcon '83*, pages 66–70, San Francisco, March 1983. IEEE.

[Tan81] A. S. Tanenbaum. *Computer Networks*, pages 148–164. Prentice-Hall, Englewood Cliffs, N.J., 1981.