Log-Based Recovery in Asynchronous Distributed Systems

Kenneth P. Kane

Ph.D Thesis

TR 89-1067
December 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

LOG-BASED RECOVERY IN ASYNCHRONOUS DISTRIBUTED SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Kenneth Paul Kane

January 1990

LOG-BASED RECOVERY IN ASYNCHRONOUS DISTRIBUTED SYSTEMS

Kenneth Paul Kane, Ph.D.

Cornell University 1990

Replication has been shown to be an important tool in the design of high-performance and highly-available distributed systems. When applied to data, however, replication significantly complicates the problem of maintaining consistency within a system. This problem is further complicated when repositories of the data can potentially fail and recover. In this dissertation, we describe a log-based mechanism for restoring consistent states to replicated data objects after failures.

A variety of techniques have been proposed for implementing consistency in a system. Most of these techniques focus on preserving a form of consistency based on serialization of updates. Although serializable consistency is useful for building a large number of applications, there are also many applications that do not require the full strength of consistency that serializability provides. For these applications, the cost of implementing serializable consistency can be prohibitive. A number of weaker and less expensive consistency forms have therefore been proposed for building such applications.

In this dissertation we focus on preserving a causal form of consistency based on the notion of virtual time. Causal consistency has been shown to apply to a variety of applications, including distributed simulation, task decomposition, and mail delivery systems. Several mechanisms have been proposed for implementing causally consistent recovery, most notably those of Strom and Yemini, and Johnson and Zwaenepoel. Our mechanism differs from these in two major respects. First, we implement a roll-forward style of recovery. A functioning process is never required to roll-back its state in order to achieve consistency with a recovering process. Second, our mechanism does not require any explicit information about the causal dependencies between updates. Instead, all necessary dependency information is inferred from the orders in which updates are logged by the object servers.

Our basic recovery technique appears to be applicable to forms of consistency other than causal consistency. In particular, we show how our recovery technique can be modified to support an atomic form of consistency that we call grouping consistency. By combining grouping consistency with causal consistency, it may even be possible to implement serializable consistency within our mechanism.

Biographical Sketch

Kenneth Paul Kane was born on June 10, 1962 at Saint Francis hospital in Wilmington, Delaware. Like all normal children, Ken always aspired to play professional football, baseball, or tennis while growing up. However, Ken's two left feet quickly got in the way of his athletic career. Ken therefore turned his attention to academics. In September 1980, Ken enrolled at the University of Delaware, where in June 1984 he received his Bachelor of Science, Magna Cum Laude, in Computer and Information Science. Because the thought of earning an honest living didn't appeal to Ken, he decided to use his education to enter graduate school. In August 1984, Ken enrolled at Cornell University to pursue his Doctorate in Computer Science. Through careful nurturing and a balanced life, Ken has grown into the well-adjusted adult he is today.

To my family.

Acknowledgements

First and foremost I would like to thank my family for all of their love and support, not only through the work on this dissertation, but throughout my life. I couldn't have done it without you.

I would also like to thank all of my friends at Cornell for making my stay there so pleasant. I would especially like to thank my officemates, Alex Aiken, Charles Elkan, Hjalli Hafsteinsson, Bruce Hendrickson, Andy Meltzer, Michael Schwartzbach, Kim Taylor, and Carolyn Turbyfill for always keeping me laughing, even when you didn't intend it. And a special thanks to Brad Vander Zanden for being such a good friend and for being the blackboard on which I derived many of the ideas in this dissertation.

I wish to express my gratitude to Prakash Panangaden and Joe Mitchell for serving on my committee, and especially to my advisor, Ken Birman, for all of his guidance during my years at Cornell.

Finally, I wish to acknowledge the generous financial support of DARPA under grant numbers N00140-87-C-8904 and NAG 2-593, and the Siemens Corporation, which allowed me to pursue this work.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Replication is an important concept in the design of fault-tolerant distributed computing systems. When applied to object-oriented systems, replication can increase the availability as well as the performance of data objects. However, replication also introduces the problem of maintaining consistency between object replicas. This problem is further compounded when object replicas can fail and recover. In this dissertation we present a recovery mechanism for restoring object replicas to consistent states after failures.

1.1 Objects and Recovery

In the last several years, object-oriented systems have become increasingly popular [HMSC88,JLHB87,LCJS87]. These systems provide their users with tools for building and maintaining abstract data objects. An object in such a system generally consists of an implementation body along with an interface. Only the interface is visible to a client of the object; implementation details such as data structures and internal procedures are hidden from the client inside the object body. Figure 1.1 depicts an object-oriented system containing two objects, a name manager and a resource allocation manager, and three clients. Clients begin by registering themselves with the name manager and then proceed to

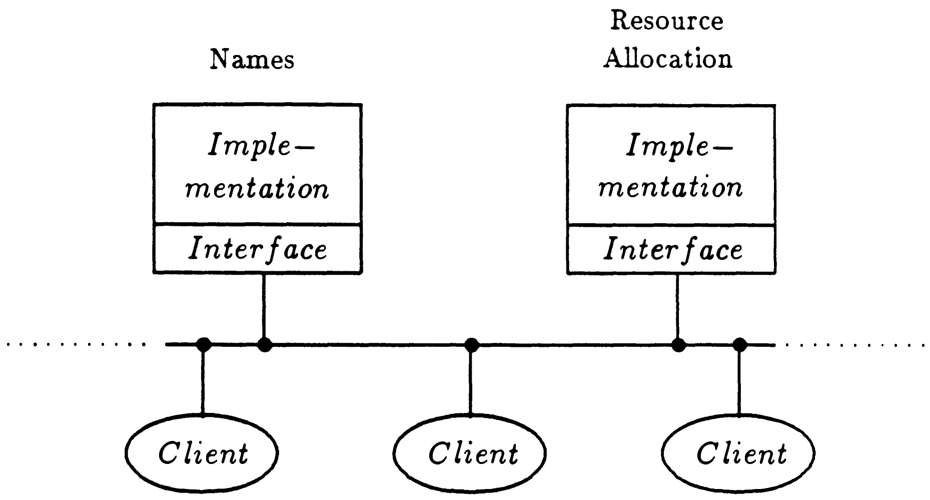


Figure 1.1: An object-oriented system

allocate resources under that name using the resource allocation manager.

Objects in a system do not necessarily exist independent of one another. The states of different objects may be related. In the above example, the state of the resource allocation manager is dependent on the state of the name manager; resources are only allocated to registered clients. When failures occur, however, consistency constraints between objects can be violated. If the name manager fails and subsequently recovers, losing some client registrations in the process, the system could reflect resources allocations to unregistered clients.

It is the purpose of this dissertation to present an automatic mechanism for restoring consistent states to (replicated) objects after failures. The mechanism is based on logging the sequences of updates that occur to object replicas and then using those sequences to construct consistent states after failures.

1.2 Consistency

The meaning of consistency in a system depends upon the application being implemented. Serializability is perhaps the most widely applied form of consistency [BG81,Gra78,Ull82]. Under serializability, operations on objects are grouped into *transactions*. Each transaction is executed as if it were an atomic unit. If a failure occurs during a transaction, the result of the transaction is as if either all of the operations in the transaction occurred or none of the operations occurred. Further, concurrent transactions are executed as if they occurred in some serial order (in reality, the operations in different transactions might be interleaved).

Serializability provides a strong consistency condition that is sufficient to guarantee correctness in large number of applications. However, for many applications the cost of implementing serializability is prohibitive. In addition, serializability often provides a stronger consistency constraint than is required by the application. For these reasons, weaker forms of consistency that are less expensive to implement have been examined.

In this dissertation we focus on a causal form of consistency based on Lamport's "*happens before*" relation [Lam78]. Under causal consistency, operations on objects are partially ordered according to the virtual time at which they occurred [Jef85] or the potential flow of information between them [BJ87a]. Objects may then only be accessed in a manner consistent with this partial ordering.

Compared with serializability, causal consistency has the advantage that it is inexpensive to implement (causally consistent message ordering can be achieved using only a one-phase protocol [BJ87b,Sch88,PBS89]). Further, causal consistency has been shown to be applicable to a large variety of applications, including mail handling systems [CP86], distributed simulation [J+87], and task decomposition [BJ87a].

1.3 Objectives

Recovery mechanisms have been proposed elsewhere for achieving causal consistency in a system [JZ88,SY85]. These mechanisms all require access to explicit information about the causal dependencies between requests. It is the goal of this work to show that consistency can be achieved without any such explicit information. Instead, consistency is achieved using only information inferred from the normal behavior of the system.

In addition, our mechanism implements a rollforward style of recovery. Many existing solutions use rollback as a synchronization technique. However, it is not always possible to rollback the state of a process or object. For example, the state of an airline reservation system reflects tickets sold to customers and money collected from those customers. If a failure occurs, rollback can be used to achieve consistency within the internal system state, but is likely to leave the state of the system inconsistent with the external world. In the airline reservation example, it would be difficult to rollback or undo ticket sales to actual customers. For this reason, our solution does not require a functioning object server to rollback its state in order to achieve consistency with a newly recovering server. This is accomplished at the cost of potentially blocking a server during its recovery.

1.4 Outline

We begin in chapter 2 by presenting our formal system model, including a description of log-based recovery and its relationship to causal consistency.

Chapter 3 then describes several consistency problems that can arise due to failures and outlines our basic recovery algorithms for solving these problems.

In chapter 4 we present transformations for consistently adding and deleting entries from server logs. These transformations are used in chapter 5 to construct solutions for the recovery problems introduced in chapter 3.

When explicit dependency information is not available in a system, our recovery algorithms can instead use dependency estimates in order to achieve consistency. These estimates must have the property that they never under-estimate the true set of dependencies. Chapter 6 presents several dependency estimates with this property. The estimates are divided into two classes: *basic* and *compound.* The compound estimates are more accurate than the basic estimates, but are also more expensive to compute.

In chapter 7 we discuss several issues concerning the efficiency of the recovery algorithms. We begin by discussing a cyclic condition that can lead to blocking during recovery. We show how this condition can be avoided by properly structuring a system. We then describe a special class of systems that can be efficiently recovered using the basic estimates, without the possibility of blocking. We conclude the chapter by outlining the problems involved in implementing object checkpoints.

Our basic recovery technique can be applied to forms of consistency other than causal consistency. In chapter 8 we describe how the recovery mechanism can be modified to provide an atomic form of consistency called *grouping* consistency.

Chapter 9 concludes the dissertation by summarizing the results and discussing several related areas for future research.

Chapter 2

Formal System Model

In this chapter we present a *partially replicated* variant of the client-server model of computation [BJ87a,BN84,Coo85]. The model is designed to represent a highly asynchronous system and focuses on those aspects of the system that are relevant to the recovery of data after a failure. The model uses *asynchronously generated logs* to record changes to data and to recover the data after failures. In addition, we describe notions of correctness and consistency based on *causality* (or which events precede others [Lam78]) and discuss their relationship to log-based recovery.

2.1 Clients and Servers

The active entities in a system are *servers* and *clients*. Servers replicate and maintain data objects that are read and updated by the clients. We let $\mathcal{SERV}$ denote the set of servers in the system and let $\mathcal{OBJS}$ denote the set of data objects managed by the servers. Each object, $A \in \mathcal{OBJS}$, is replicated at some subset of the servers, $\mathcal{SERV}_A$, which we refer to as the *server set* of the object ($\mathcal{SERV}_A \subseteq \mathcal{SERV}$). For convenience, we will denote the set of objects managed by a server, f, as $\mathcal{OBJS}_f$.

$$\mathcal{OBJS}_f = \{ A \in \mathcal{OBJS} \mid f \in \mathcal{SERV}_A \}$$

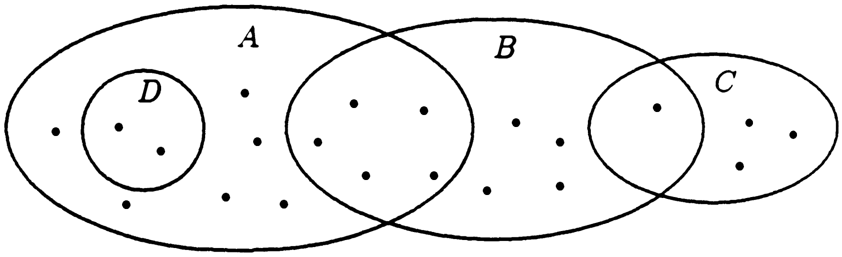


Figure 2.1: Overlap between object server sets

Figure 2.1 illustrates the overlap between the server sets of different objects in an example system. Depicted are the server sets of four objects: A, B, C, and D. Note that the server set of object D is completely contained within the server set of object A.

A client accesses (reads or updates) an object by broadcasting its request to all servers managing a replica of the object. Upon receiving a request, each server makes the appropriate update to its object replica. We assume that the state of a replica is completely determined by the sequence of updates received by the replica's server and that other factors, such as the time of an update's receipt or the timing between updates, do not affect a replica's state. It is not necessary, however, that all servers receive requests in the same order. Concurrently issued requests can be received by different servers in different orders, provided that those orders lead to equivalent object states. This issue is discussed in further detail in section 2.2.

As an example, consider a system service for managing lists. This service might provide users with functions for creating new lists, adding and deleting entries from existing lists, and querying the contents of lists. One use for such a service would be to manage resource allocations to client processes. Clients would begin by submitting their names to a list of registered processes. Once registered, clients could allocate resources by making entries into a resource allocation list. Such a system is depicted in figures 2.2 and 2.3. In both figures, the list of registered process names is replicated at servers f and g, while the list of allocated resources is replicated at servers g and h. Figure 2.2 depicts the concurrent submission of two client name registration messages (reg_1 and reg_2). Figure 2.3 depicts the concurrent submission of two resource allocation messages (alc_1 and alc_2). Note that in both examples the concurrent submissions are received in different orders by the servers.

It may seem unusual that a server may manage replicas of multiple objects. However, in object-oriented systems that replicate data, we believe that such overlap between the server sets of objects is common. The work in dissertation was motivated by the need to implement failure recovery in the ISIS system [BCJ+]. In the ISIS system, servers often implement general objects, such as list management in the previous example. These objects are then used by clients to implement more specific services, such as name management and resource allocation. Because of availability and performance considerations, not all of the general servers may manage each of the specific services. Further, the subset of servers that do manage a specific service may dynamically change as servers fail and recover, or as different availability and performance constraints are placed on the service. As a result, general object servers often manage multiple specific services.

2.2 Request Ordering and Causality

Clients in a system interact with each other in many ways. Clients communicate directly by sending messages to one another, and indirectly through the objects managed by the servers. These interactions may lead to causal dependencies between the object requests they invoke. For example, in the system of figure 2.3, two clients may agree to transfer an allocated resource between them. When this occurs, the allocation service is notified of the transfer through a re-allocation

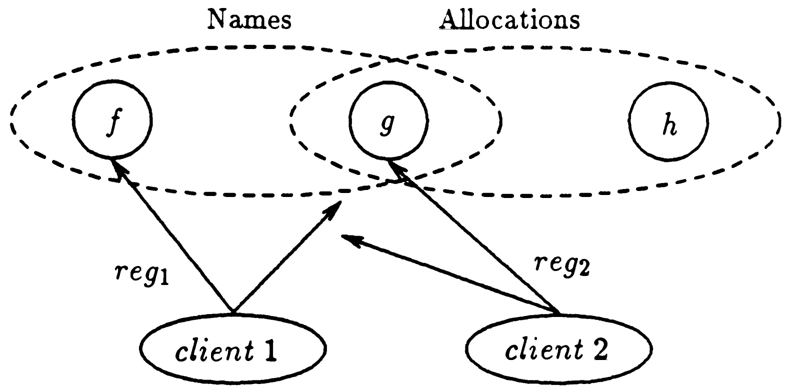


Figure 2.2: Concurrent submission of two name registration messages

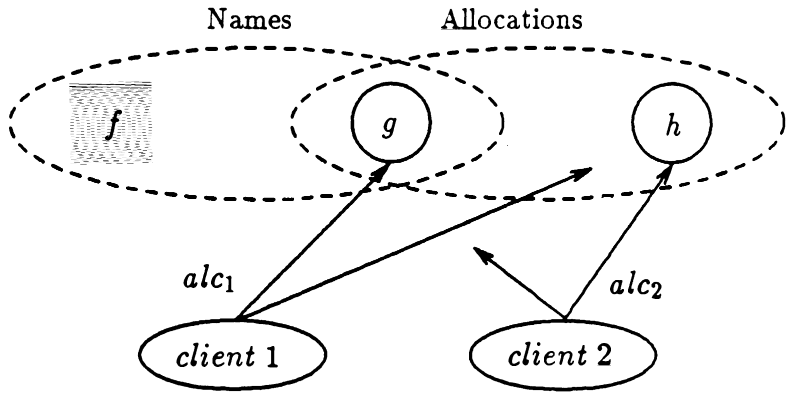


Figure 2.3: Concurrent submission of two resource allocation messages

$$\text{Request Structure: } (\mathcal{R}, \prec_{\mathcal{R}})$$
$$\mathcal{R} = \{reg_1,\ reg_2,\ alc_1,\ alc_2\}$$
$$reg_1 \prec_{\mathcal{R}} alc_1 \qquad reg_2 \prec_{\mathcal{R}} alc_2$$

Figure 2.4: Resource allocation request structure

request message sent by the clients. This re-allocation request is causally dependent on the original allocation request (as well as on the registration requests of the clients involved); no server should receive the transfer request until it has received the clients' registration messages and the resource's initial allocation message.

We summarize the set of causal dependencies between the client requests in a system by means of a *request structure.* A request structure is a logical entity designed to represent the behavior of clients as seen by an outside observer looking back on the system after its completion. As such, the request structure of a system is static.

Definition 2.1

A request structure is a partially ordered set of requests $(\mathcal{R}, \prec_{\mathcal{R}})$.

Here, $\mathcal{R}$ is the set of all requests made by clients in the system and $\prec_{\mathcal{R}}$ relates all pairs of causally dependent requests. If two requests are related, $x \prec_{\mathcal{R}} y$, then request y is causally dependent on request x. The relation $\prec_{\mathcal{R}}$ is equivalent to the "*happens before*" relation of Lamport [Lam78] and like the "happens before" relation $\prec_{\mathcal{R}}$ is transitive and acyclic. $\mathcal{R}$ may contain requests made on many different objects. For any request, $x \in \mathcal{R}$, we will sometimes use the notation $x.A$ to indicate that request x was made on object A. A request structure representing the dependencies in the resource allocation system is shown in figure 2.4.

Recall that servers process requests in the order in which they receive them.

We assume that in order to construct correct replica states, servers must receive (process) requests in causally consistent orders (*i.e.* in orders consistent with the application's request structure $(\mathcal{R}, \prec_{\mathcal{R}})$). If a server receives two related (ordered) requests, $x \prec_{\mathcal{R}} y$, then it must receive request x before it receives request y. Unrelated requests may be received by a server in any order and different servers may even receive the same unrelated requests in different orders.

We do not assume that servers are given any explicit information about the dependencies between the requests they receive. In particular, we do not assume that servers have any explicit knowledge of $(\mathcal{R}, \prec_{\mathcal{R}})$. It is the responsibility of the clients to ensure that all servers perceive causally consistent request orderings. A variety of techniques exist for clients to order their requests [BJ87b, CM84,CASD86,PBS89]. We will not, however, make any assumption about the mechanism used. Clients may use any technique that guarantees correct request orderings.

2.3 Failures and Recovery

We assume *fail-stop* servers [SS83]. When a server fails, it immediately ceases to receive and process client requests, and the other servers in the system are notified of its failure. In addition, the failed process also loses the contents of its volatile memory. We assume that other types of failures, such as send/receive omission failures [PT86] or Byzantine (malicious) failures [LSP82], do not occur. We also assume that network partitions [DGMS85] never occur, so that non-failed servers can always communicate between themselves.

In order to support recovery from failures, each server maintains a log of the object updates it performs.

Definition 2.2

A log is a totally ordered set of requests $(\mathcal{L}, \rightarrow_{\mathcal{L}})$.

Here, $\mathcal{L}$ is the set of object update requests received by the server and $\rightarrow_{\mathcal{L}}$ is their order within the log. For the present, logs will be restricted to contain only requests; they will not contain checkpoints. In any real system checkpoints are necessary to limit the growth of logs. However, the presence of checkpoints complicates the problem of recovery and so their use will be postponed until chapter 7.

Servers log requests in the order in which they receive them. Because servers receive requests in causally consistent orders, it follows that servers log requests in orders consistent with the application's request structure.

Definition 2.3

The log, $(\mathcal{L}_f, \rightarrow_f)$, of a server f is <u>consistent</u> with a request structure, $(\mathcal{R}, \prec_{\mathcal{R}})$, if

1. $\forall\, x.A \in \mathcal{L}_f :\ f \in \mathcal{SERV}_A$
2. $\forall\, x.A \in \mathcal{L}_f :\ \forall y.B \in \mathcal{R} :$
$$(y.B \prec_{\mathcal{R}} x.A\ \wedge\ f \in \mathcal{SERV}_B) \Longrightarrow (y.B \in \mathcal{L}_f\ \wedge\ y.B \rightarrow_f x.A)$$

In the treatment that follows, we assume that a request is logged by a server as soon as it is received and processed, and so the log of a server always reflects the current states of the server's object replicas. For efficiency, a server could decouple its execution speed from that of its log by buffering requests in memory and periodically flushing the buffer to its log. A server's log would then reflect states that lag behind the actual states of its replicas. Managing a server's log asynchronously from its replicas does not affect the validity of our results. However, it would complicate the discussion. If it were really desired to implement this restriction, a server could use a technique such as *write-ahead logging* [BHG87].

Servers in our model do not coordinate their logs with those *of other servers.*

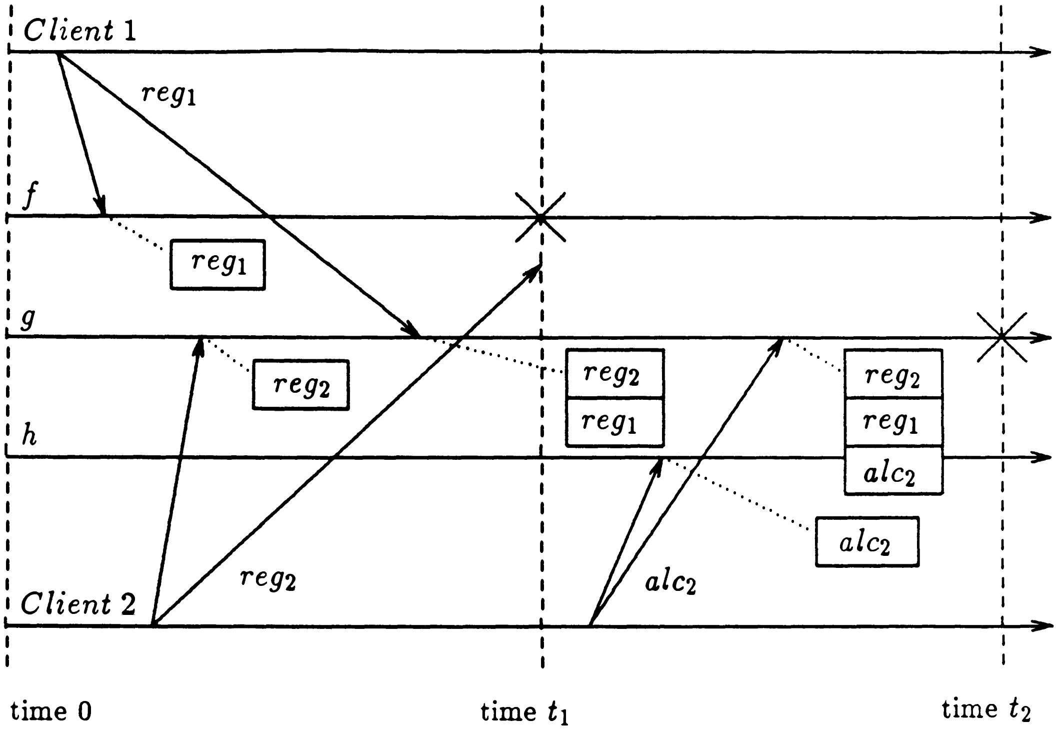


Figure 2.5: An execution of the resource allocation system

Each server logs the requests it receives independent of the times when those requests are logged by other servers. As a result, the state of an object represented in one log may fall behind the state of that object represented in some other log. Further, because servers do not always receive requests in the same order, different servers may have logged different requests for the same object at any one time.

Figure 2.5 illustrates one possible execution of the system of figures 2.2 and 2.3. In the figure, horizontal lines represent client and server executions through time while diagonal arrows represent request message broadcasts. Depicted are the broadcasts of two name registration messages (reg_1 and reg_2) and one resource allocation message (alc_2). Note that server f fails at time t_1 before receiving and logging the second registration message, and that server g fails at time t_2 after receiving and logging all three broadcasts. The contents of each server's log are shown below that server's time line after each request receipt.

Managing server logs asynchronously from one another reduces the system overhead by decoupling the execution speeds of different servers. Each server is free to process requests at a rate independent of the other servers. Unfortunately, as we will see in the next chapter, the use of asynchronous logs leads to coordination problems between servers after failures. These problems can be avoided by coordinating the logs of different servers (*pessimistic* logging techniques exist for doing this [JZ87,PP83]). However, this adds substantial overhead to the normal operation of a system. We therefore choose to manage logs asynchronously, postponing the overhead of coordinating logs until the time of a server's failure recovery. If failures are rare, this *optimistic* approach should lead to good performance of the system.

Other optimistic logging techniques have been proposed for managing failures in distributed systems [SY85,JZ88]. These techniques involve maintaining explicit information about the causal dependencies between updates. Managing such information can be difficult or impossible, though, when the set of clients is either unknown to the servers or large and dynamically changing. We therefore examine the problem of optimistic failure recovery in systems where explicit dependency information is not available.

A server uses its log to recover from failures in the usual way. In order to restore the state of a failed object replica, a recovering server simply re-executes the sequence of updates logged for the object. Once the recovering server has restored its (volatile) replica of an object, that server begins receiving, processing, and logging new requests on the object. We refer to a server that is in the process of restoring its replica of an object as a *recovering server* of that object and we refer to a server that can process new requests on an object as an *active server* of the object.

Note that a recovering server does not have to re-execute the updates for an object in the order in which they were logged. Previously we stated that a server's object replicas are correct if that server processes requests in causally consistent orders. Because of this, a recovering server can re-execute logged updates in any order consistent with the application's request structure, and still reconstruct valid object replicas. Of course, the order in which a server logs requests is always consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$, and so this order can be used to construct valid replica states. This is particularly useful when servers does not have access to any explicit dependency information, and so cannot determine other valid request orderings.

We represent the state of an object reflected in a server's log by the set of updates it contains for that object.

Definition 2.4

The projection of a log, $(\mathcal{L}_f, \rightarrow_f)$, onto an object, $A \in \mathcal{OBJS}$, is

$$(\mathcal{L}_f, \rightarrow_f) \mid_A \;=\; \{x.A \mid x.A \in \mathcal{L}_f\}$$

2.4 System State and Consistency

The state of a system can be summarized in terms of the contents of the servers logs and the status of each server (the log of an active server reflects the actual states of the server's replicas).

$$\mathcal{ACT}_{S/Names} = \emptyset \quad \mathcal{REC}_{S/Names} = \{f\} \quad \mathcal{FAIL}_{S/Names} = \{g\}$$
$$\mathcal{ACT}_{S/Allocations} = \{h\} \quad \mathcal{REC}_{S/Allocations} = \emptyset \quad \mathcal{FAIL}_{S/Allocations} = \{g\}$$

$$(\mathcal{L}_{S/f}, \rightarrow_{S/f}): \begin{array}{|c|}\hline reg_1 \\ \hline \end{array} \quad (\mathcal{L}_{S/g}, \rightarrow_{S/g}): \begin{array}{|c|}\hline reg_2 \\ \hline reg_1 \\ \hline alc_2 \\ \hline \end{array} \quad (\mathcal{L}_{S/h}, \rightarrow_{S/h}): \begin{array}{|c|}\hline alc_2 \\ \hline \end{array}$$

Figure 2.6: A possible state of the resource allocation system

Definition 2.5

A <u>*state*</u>*, S, of the system is characterized by the following values:*

For each data object, $A \in \mathcal{OBJS}$:

$\mathcal{ACT}_{S/A}$ *The set of active servers of object A.*

$\mathcal{REC}_{S/A}$ *The set of recovering servers of object A.*

$\mathcal{FAIL}_{S/A}$ *The set of failed servers of object A.*

For each server, $f \in \mathcal{SERV}$:

$(\mathcal{L}_{S/f}, \rightarrow_{S/f})$ *The log of server f.*

For example, consider again the execution of figure 2.5. Suppose that server f begins to recover at time t_2, when server g fails. In this case, figure 2.6 shows the state, S, of the system immediately after time t_2.

When a server fails, it fails for all objects it manages. When the server later recovers, it begins recovering the states of all replicas it manages.

$$(\exists\, A \in \mathcal{OBJS}:\ f \in \mathcal{FAIL}_{S/A}) \implies$$
$$(\forall\, A \in \mathcal{OBJS}_f:\ f \in \mathcal{FAIL}_{S/A})$$

We denote the complete set of failed servers in state S as $\mathcal{FAIL}_S$.

$$\mathcal{FAIL}_S = \bigcup_{A \in \mathcal{OBJS}} \mathcal{FAIL}_{S/A}$$

In this dissertation, we will be concerned with the problem of maintaining the overall consistency of a system's state (as well as the consistency of server logs) when servers fail and recover. There are two aspects to the issue of a system's overall consistency. First, there is the issue of consistency between the replicas of the same object. Second, there is the issue of consistency between the states of different objects. We briefly discuss each of these aspects in turn. A more formal treatment of these issues is reserved for chapter 3.

All active servers of an object should maintain equivalent states for their object replicas, so that the servers behave consistently with respect to one another. Because servers execute asynchronously from one another, different servers may construct this state at different speeds and by processing requests in different orders. We assume that at the time a server recovers, all active servers of an object have constructed (and logged) equivalent object states. This state, which we refer to as the active state of the object, is the state the recovering server should restore to its replica.

Definition 2.6

The active state of an object, $A \in \mathcal{OBJS}$, in system state S is

$$\mathcal{AS}_{S/A} = (\mathcal{L}_{S/f}, \rightarrow_{S/f}) \mid_A \quad \forall f \in \mathcal{ACT}_{S/A}$$

Restricting active servers to equivalent object states (at the time of a server recovery) is reasonable. For example, in the ISIS system [BJ87b] process failure and recovery events are totally ordered with respect to all other events (message broadcasts) in the system. Thus, when a server recovers from a failure, it can assume that all active servers of an object have received the same set of requests and thereby constructed the same object state. Note that the restriction on identical states is only required to hold at the time of a server recovery. At all other times during the execution of the system, servers are free to maintain their object replicas asynchronously.

The second aspect to the issue of a system's overall consistency is consistency between the states of different objects. The state of an object should never reflect a request (update) unless all of the requests on which it is causally dependent are also reflected in their object's active states. For example, a system running under the request structure of figure 2.4 should never be in a state that reflects the allocation (alc_1) made by the first client without reflecting the client's registration (reg_1).

A system state, S, is said to be observably consistent with a request structure $(\mathcal{R}, \prec_{\mathcal{R}})$, if the above consistency constraints hold within the active portion of the system. That is, a state is consistent with a request structure if all active servers of an object have logged the same (valid) state for the object and the states of all different active objects are mutually consistent. These constraints are only required to hold within the active part of a system because this is the only portion of the system visible to clients.

Definition 2.7

A system state, S, is observably consistent with a request structure, $(\mathcal{R}, \prec_{\mathcal{R}})$, if

1. $\forall\ f \in \mathcal{SERV} - \mathcal{FAIL}_S\ :\ (\mathcal{L}_{S/f}, \rightarrow_{S/f})$ *is consistent with* $(\mathcal{R}, \prec_{\mathcal{R}})$.
2. $\forall\ A \in \mathcal{OBJS} :\ \forall\ f, g \in \mathcal{ACT}_{S/A} :\ (\mathcal{L}_{S/f}, \rightarrow_{S/f})\ |_A\ =\ (\mathcal{L}_{S/g}, \rightarrow_{S/g})\ |_A$
3. $\forall\ A, B \in \mathcal{OBJS}\ (\mathcal{ACT}_{S/A} \neq \emptyset\ \wedge\ \mathcal{ACT}_{S/B} \neq \emptyset) :$
$$\forall\ x.A \in \mathcal{AS}_{S/A} :\ \forall\ y.B \in \mathcal{R}\ (y.B \prec_{\mathcal{R}} x.A) :\ y.B \in \mathcal{AS}_{S/B}$$

This dissertation presents a recovery mechanism for maintaining observable consistency in the presence of server failures and recoveries.

2.5 Summary

This chapter presented a formal model of replicated data in an asynchronous distributed system. The model was designed to focus on those aspects of the system relevant to the recovery of data after a failure.

A system consisted of a set of servers, $\mathcal{SERV}$, replicating a set of data objects, $\mathcal{OBJS}$, along with a set of clients that accessed and updated those objects. A basic assumption was that objects were partially replicated within larger groups of servers. This lead to arbitrary overlap between the sets of objects individual servers managed. A client in the system accessed an object by broadcasting a request message to all servers of the object. An underlying structure, $(\mathcal{R}, \prec_{\mathcal{R}})$, governed the correct orders in which servers could receive requests. Because this request structure was unknown to the servers, it was the responsibility of the clients to ensure the servers perceived correct message orderings.

In order to support recovery from fail-stop failures, each server maintained a log, $(\mathcal{L}_f, \rightarrow_f)$, of the client requests it received. There was no synchronization between the logs of different servers. Each server logged requests as soon as they were received. It was noted that the order in which requests appear within logs is always consistent with the application's request structure. After a failure, a server reconstructed the states of its object replicas by replaying the requests in its log.

Servers could recover differing replica states because logs were maintained asynchronously. A system was said to be observably consistent if three conditions held:

1. The order of requests in all servers' logs (*i.e.* the states of the servers' replicas) are consistent with the application's request structure.

2. All active servers of an object have logged (constructed) the same state for the object.

3. The states of all active objects are mutually consistent (*i.e.* consistent with respect to the application's request structure.

Developing a recovery mechanism for maintaining this consistency is the goal of this dissertation.

Chapter 3

Consistency Problems

The use of asynchronous logs potentially allows servers to recover inconsistent states after failures. This chapter describes (in outline form) a recovery mechanism for preventing such inconsistencies. The chapter begins by presenting several examples of how inconsistencies arise. The behavior of the recovery mechanism is then formally described and several examples of its operation are given. This chapter presents only a formal outline of the recovery mechanism. The implementation of the mechanism is the subject of the remainder of this dissertation.

3.1 Problem Examples

Two types of inconsistencies can develop in a system: those between the states of an object's different replicas and those between the states of different objects. We present three examples of such inconsistencies. The first two illustrate inconsistencies that can develop between an object's replicas. The last illustrates an inconsistency that can develop between the states of two objects.

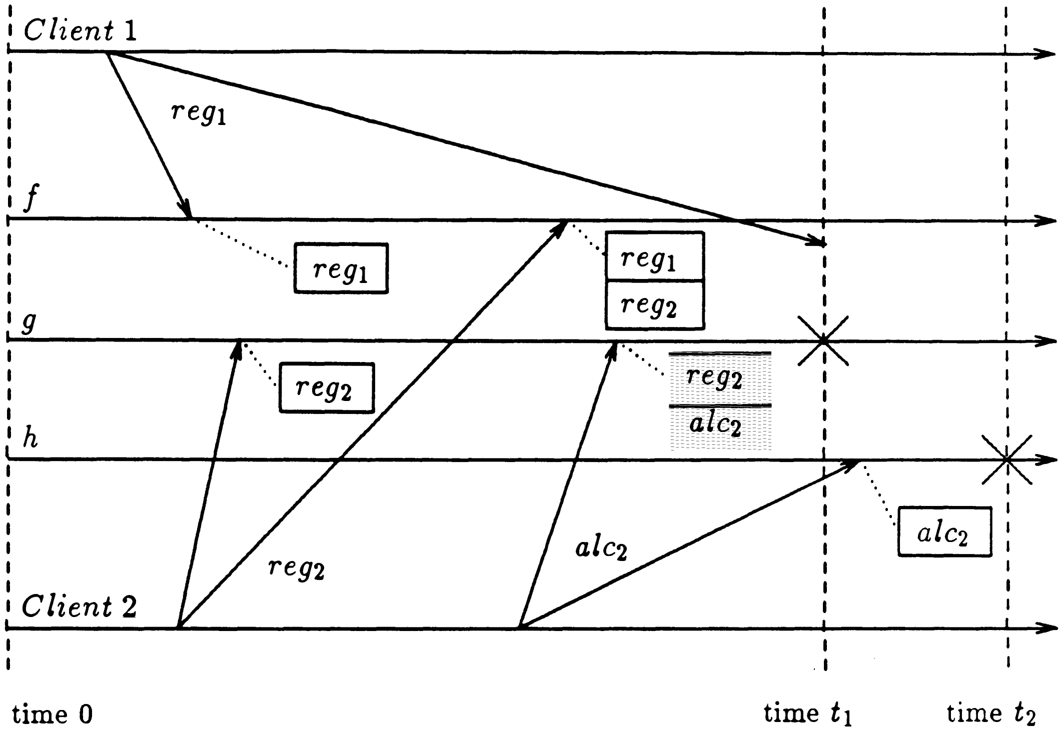


Figure 3.1: Inconsistency with an active replica

3.1.1 Consistency with Active Replicas

At the time a server recovers from a failure, its log reflects the states of its object replicas from the time of the failure. When the recovering server replays its log, it restores its replicas into these states. These states may, however, be out of date if other servers of the objects remained active, processing updates after the recovering server's failure. Such updates would be reflected in the replicas of the active servers, but not in the replicas of the recovering server.

For example, consider the execution of the resource allocation system shown in figure 3.1. The execution depicts the transmission of two client registration messages (reg_1 and reg_2) and one resource allocation request (alc_2). In the figure, server f receives both registration requests without failing. Server g fails at time t_1 after receiving requests reg_2 and alc_2, but before receiving request reg_1. And, server h fails after receiving request alc_2. Suppose that server g recovers after time t_1. Server g will then recover its replica of object "Names" into a state reflecting only the registration of client 2. It will not recover the registration of client 1 reflected in the object's active state (the state reflected in the replica of server f). The contents of both servers logs at the time of the recovery are shown below:

Server f (active) :	Server g (recovering) :
reg_1	reg_2
reg_2	alc_2

This type of inconsistency can be prevented by transferring the active states of objects to the failed server at the time of recovery. The recovering server would then alter its log to reflect these transferred states so that it restores them during log replay. This is the approach used by ISIS [BJ87a] and will be the approach used in our recovery mechanism.

3.1.2 Consistency between Recovering Replicas

A similar type of inconsistency can occur when several servers of an inactive object (an object for which all servers have failed) recover simultaneously. Because the servers maintain their logs asynchronously from one another, and because they probably failed at different times, each server's log probably reflects a different state of the object. Each server is therefore likely to recover a state for its object replica that differs from (is inconsistent with) the states recovered by the other servers.

For example, consider the execution of the resource allocation system shown in figure 3.2. This execution is similar to the previous one except that server f fails before receiving registration request reg_2. Suppose that both servers f and g simultaneously recover at some point after time t_2. The servers will then recover inconsistent states for their replicas of "Names". Server f will recover a

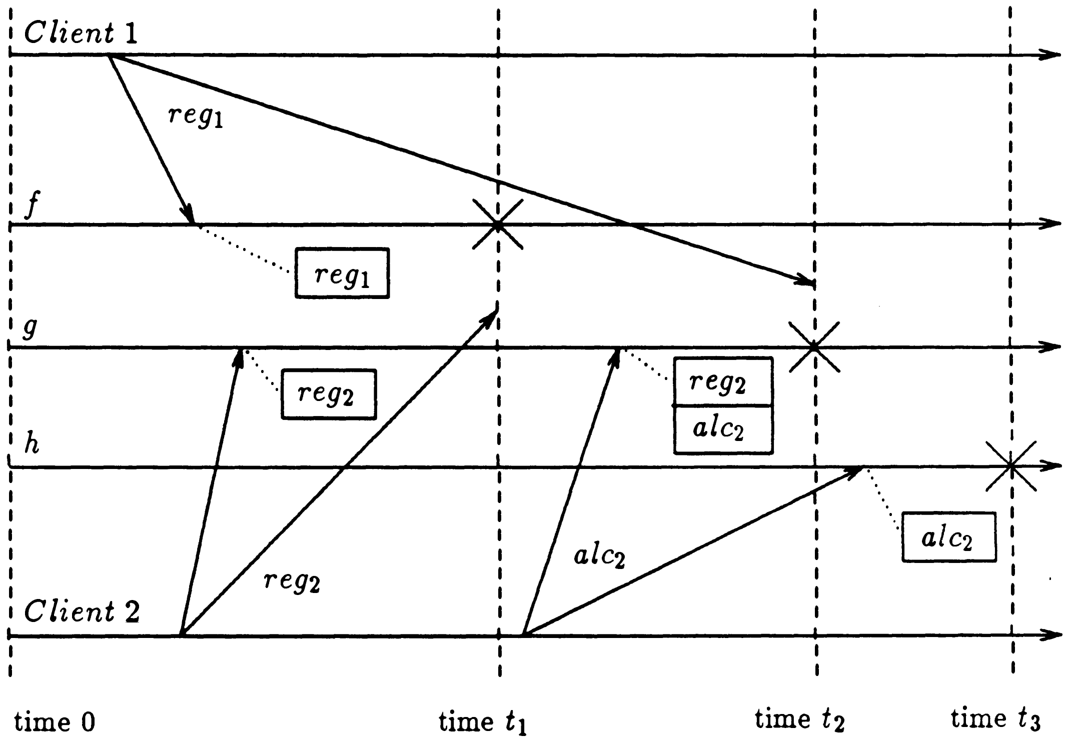


Figure 3.2: Inconsistency between recovering replicas

state reflecting only the registration of client 1 and server g will recover a state reflecting only the registration of client 2. This situation is depicted below:

$$\begin{array}{c} \text{Server } f \\ \text{(recovering)} \end{array} : \boxed{reg_1} \qquad \begin{array}{c} \text{Server } g \\ \text{(recovering)} \end{array} : \begin{array}{|c|} \hline reg_2 \\ \hline alc_2 \\ \hline \end{array}$$

This inconsistency problem can be solved by having the recovering servers choose a new state for the object and then alter their logs so that they all recover this state during log replay. Ideally, this state should be a recent one, reflecting as many of the client requests as possible. In synchronous systems, where the logs of servers are coordinated, the log of the last server to fail [Ske85] will contain the most recent state of the object. This state could then be used to recover the failed servers. When logs are not coordinated, however, any server may have logged the most recent state. Different servers may even have logged different sets of requests and so no server will have logged the most recent state. In this case, a recent state of the object can be formed by merging the logged requests of the recovering servers. This is the approach used by our recovery mechanism.

3.1.3 Consistency between Active Objects

The previous two examples illustrated consistency problems that develop between different replicas of a single object. Because dependencies can exist between requests on different objects, inconsistencies can also develop between the states of different objects. Let S denote a state of a system in which some failed server f is recovering its replica of an object, A, and in which some other object, B, is active. If the state of object A logged by server f is old, f may recover a state that does not reflect all of the updates on which the active state of B ($\mathcal{AS}_{S/B}$) depends. Similarly, if the active state of B is old (*i.e.* it is the result of a previous failure recovery of its servers), it may be missing updates on which the state of A recovered by server f depends.

As an example, consider again the execution shown in figure 3.2. If servers f and h recover at some point after time t_3, they will recover mutually inconsistent states. Server h will recover an allocation request (alc_2) from a client whose registration (reg_2) is not recovered by server f. That is, the servers will recover a state that reflects a client's allocation without reflecting the registration on which it depends. Shown below are the logs of the two servers at the time of the recovery:

$$\begin{matrix}\text{Server } f \\ \text{(recovering)}\end{matrix} : \boxed{reg_1} \qquad \begin{matrix}\text{Server } h \\ \text{(recovering)}\end{matrix} : \boxed{alc_2}$$

Inconsistencies between different objects are the most difficult ones to prevent in a system, and are the focus of the recovery mechanism.

3.2 Recovery Mechanism

In order to preserve consistency within a system, a recovering server must be careful about the states it restores to its object replicas. A recovering server must restore replicas of active objects using those objects' current states. A recovering server must also restore replicas of inactive objects to states consistent with the rest of the system (*e.g.* the state must agree with those of other recovering replicas of the object, and the state must be consistent with the states of other active objects in the system).

Our recovery mechanism enforces these constraints in two phases. In the first phase, a failed server's replicas of active objects are restored to the objects' current states in the system. We refer to this as the server's JOIN phase. Once the server has completed its JOIN phase, its replicas of inactive objects are restored to states consistent with the state of the system. We refer to this as the server's ACTIVATE phase. Figure 3.3 illustrates the relationship of the two recovery phases. The behaviors of the two phases are formally outlined in the following sections.

JOIN Phase: (immediately upon recovery)

1. **for each** $A \in \mathcal{OBJS}_f$ $(\mathcal{ACT}_{S/A} \neq \emptyset)$

   alter $(\mathcal{L}_{S/f}, \rightarrow_{S/f})$ so that

   $$(\mathcal{L}_{S/f}, \rightarrow_{S/f}) \mid_A \; = \; \mathcal{AS}_{S/A}$$

2. reconstruct replicas of active objects from $(\mathcal{L}_{S/f}, \rightarrow_{S/f})$
3. begin processing new requests on active objects

ACTIVATE Phase: (upon completion of JOIN phase)

4. while $\exists A \in \mathcal{OBJS}_f$ $(\mathcal{ACT}_{S/A} = \emptyset)$

   wait for all $g \in \mathcal{REC}_{S/A}$ to complete their JOIN phases

   construct a new state, S_A, for object A by merging the logs
   of all members of $\mathcal{REC}_{S/A}$

   if S_A is inconsistent with the state of any active object
   then abort activation of A until additional servers
   recover

   activate object A by:

   altering $(\mathcal{L}_{S/f}, \rightarrow_{S/f})$ so that

   $$(\mathcal{L}_{S/f}, \rightarrow_{S/f}) \mid_A \; = \; S_A$$

   reconstruct replica of A from $(\mathcal{L}_{S/f}, \rightarrow_{S/f})$

   begin processing new requests on A

Figure 3.3: Recovery sequence of server f in state S

The recovery sequence of a server is divided into two phases for several reasons. The JOIN phase provides a server with information about the states of some of the active objects in the system. This information is used in the ACTIVATE phase to ensure that only consistent states are recovered for inactive objects. A consistent state cannot always be recovered, however, for an inactive object; moreover, the ACTIVATE phase cannot always determine (based on the dependency information available to it) if the state it constructed for an object is consistent with the states of all active objects. When it cannot determine the consistency of a state, the ACTIVATE phase must temporarily abort the recovery of an object until other servers recover, providing additional dependency information. The JOIN phase, on the other hand, never needs to abort and so it is separated from the ACTIVATE phase.

3.2.1 JOIN Phase Outline

When a server begins recovering from a failure, its status is upgraded from a failed server to a recovering server for each object it manages. The JOIN phase is responsible for bringing the state of a newly recovering server up to date with respect to the states of active objects in the system. The current states of active objects are transferred from the active servers to the recovering server and the recovering server's log is altered to reflect these current object states. The recovering server's replicas are then restored by replaying the appropriate portion of the log and the server begins processing new client requests on the objects.

The changes that occur to the system state as a result of the JOIN phase are summarized in definition 3.1. Note that the only portion of the state that changes is the portion related to the recovering server (f).

Definition 3.1

A state, T, solves the JOIN problem for server $f \in \mathcal{SERV}$ in state S under request structure $(\mathcal{R}, \prec_{\mathcal{R}})$ if T satisfies the following conditions:

JC1. $(\mathcal{L}_{T/f}, \rightarrow_{T/f})$ *is consistent with* $(\mathcal{R}, \prec_{\mathcal{R}})$.

JC2. *The new log of server f reflects the current states of active objects.*

$$\forall\, A \in \mathcal{OBJS}_f\ (\mathcal{ACT}_{S/A} \neq \emptyset):\quad (\mathcal{L}_{T/f}, \rightarrow_{T/f})\,|_A \;=\; \mathcal{AS}_{S/A}$$

JC3. *The only log that changes is that of server f.*

$$\forall\, g \in \mathcal{SERV}\ (g \neq f):\quad (\mathcal{L}_{T/g}, \rightarrow_{T/g}) \;=\; (\mathcal{L}_{S/g}, \rightarrow_{S/g})$$

JC4. *Server f changes from a recovering to an active server of the active objects.*

$$\begin{aligned}
&\forall\, A \in \mathcal{OBJS}\ (\mathcal{ACT}_{S/A} \neq \emptyset):\\
&\quad f \in \mathcal{SERV}_A \implies\\
&\qquad (\mathcal{ACT}_{T/A} = \mathcal{ACT}_{S/A} \cup \{f\} \ \wedge\ \mathcal{REC}_{T/A} = \mathcal{REC}_{S/A} - \{f\})\\
&\quad f \notin \mathcal{SERV}_A \implies\\
&\qquad (\mathcal{ACT}_{T/A} = \mathcal{ACT}_{S/A} \ \wedge\ \mathcal{REC}_{T/A} = \mathcal{REC}_{S/A})\\
&\forall\, A \in \mathcal{OBJS}\ (\mathcal{ACT}_{S/A} = \emptyset):\\
&\quad (\mathcal{ACT}_{T/A} = \mathcal{ACT}_{S/A} \ \wedge\ \mathcal{REC}_{T/A} = \mathcal{REC}_{S/A})
\end{aligned}$$

JC5. *The set of failed servers remains the same.*

$$\forall\, A \in \mathcal{OBJS}:\ \mathcal{FAIL}_{T/A} = \mathcal{FAIL}_{S/A}$$

In addition to meeting these conditions, the new log of server f should also be as complete as possible. The new log should retain as many of the old log's entries as possible. This allows the ACTIVATE phase to recover inactive objects into the most recent state possible. Although we will not formalize this condition, we do wish to point it out as a goal.

As shown in the following theorem, the JOIN phase preserves consistency within a system.

Theorem 3.1

If S is a state that is observably consistent with a request structure $(\mathcal{R}, \prec_{\mathcal{R}})$, and if T is a state that solves the JOIN problem for server $f \in \mathcal{SERV}$ in state S, then T is also observably consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$.

Proof: In order to prove that T is observably consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$ we must show three things. First, we must show that all servers' logs are consistent with the request structure. From condition JC1 of the JOIN phase definition we know that the log of server f (in state T) is consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. From condition JC3 we know that the logs of all other servers remain unchanged from state S, in which they were all consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$ by premise. The logs of all servers in state T are therefore consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$.

Next, we must show that all active servers of an object reflect the same state for the object. Let $A \in \mathcal{OBJS}$ be any active object (*i.e.* $\mathcal{ACT}_{T/A} \neq \emptyset$). We assume that f is not actively servering object A in state S (*i.e.* $f \notin \mathcal{ACT}_{S/A}$), otherwise it would not need to solve its JOIN problem. By premise, S is an observably consistent state and so all active servers of A in S have logged the same object state.

$$\forall\, g \in \mathcal{ACT}_{S/A} : \ (\mathcal{L}_{S/g}, \rightarrow_{S/g}) \mid_A \ = \ \mathcal{AS}_{S/A}$$

Because $f \notin \mathcal{ACT}_{S/A}$, it follows from condition JC3 that the logs of all servers in $\mathcal{ACT}_{S/A}$ remain unchanged between states S and T.

$$\forall\, g \in \mathcal{ACT}_{S/A} : \ (\mathcal{L}_{T/g}, \rightarrow_{T/g}) \ = \ (\mathcal{L}_{S/g}, \rightarrow_{S/g})$$

Combining these two equations we see that all active servers of A in state S have still logged the same object state in state T.

$$\forall\, g \in \mathcal{ACT}_{S/A} : \; (\mathcal{L}_{T/g}, \rightarrow_{T/g}) \mid_A \;=\; \mathcal{AS}_{S/A} \tag{3.1}$$

Now, there are two cases: either f is a (recovering) server of A or it is not. Suppose f is a server of A. From condition JC2 we know that

$$(\mathcal{L}_{T/f}, \rightarrow_{T/f}) \mid_A \;=\; \mathcal{AS}_{S/A} \tag{3.2}$$

Combining equations 3.1 and 3.2 we get

$$\forall\, g \;\in\; \mathcal{ACT}_{S/A} \bigcup \{f\} \;:\; (\mathcal{L}_{T/g}, \rightarrow_{T/g}) \mid_A \;=\; \mathcal{AS}_{S/A} \tag{3.3}$$

From condition JC4 we know that

$$\mathcal{ACT}_{T/A} \;=\; \mathcal{ACT}_{S/A} \bigcup \{f\}$$

Substituting this into equation 3.3 we get the desired result that all active servers of A in state T reflect the same state for the object.

$$\forall\, g \in \mathcal{ACT}_{T/A} : \; (\mathcal{L}_{T/g}, \rightarrow_{T/g}) \mid_A \;=\; \mathcal{AS}_{S/A} \tag{3.4}$$

Now suppose that f is not a server of object A. From condition JC4 we know that $\mathcal{ACT}_{T/A} = \mathcal{ACT}_{S/A}$. Substituting this into equation 3.1 we see again that all servers of A are consistent.

$$\forall\, g \in \mathcal{ACT}_{T/A} : \; (\mathcal{L}_{T/g}, \rightarrow_{T/g}) \mid_A \;=\; \mathcal{AS}_{S/A} \tag{3.5}$$

The last thing we must show in order to prove the observable consistency of T is that the states of all active objects are mutually consistent. Because S is an observably consistent state we know that all active objects are mutually consistent in state S.

$$\begin{aligned} &\forall\, A, B \in \mathcal{OBJS}\; (\mathcal{ACT}_{S/A} \neq \emptyset \;\wedge\; \mathcal{ACT}_{S/B} \neq \emptyset) : \\ &\qquad \forall\, x.A \in \mathcal{AS}_{S/A} : \; \forall\, y.B \in \mathcal{R}\; (y.B \prec_{\mathcal{R}} x.A) : \; y.B \in \mathcal{AS}_{S/B} \end{aligned} \tag{3.6}$$

From condition JC4 it follows that any object that is active in state S is also active in state T and that there are no new active objects in state T.

$$\forall\ A \in \mathcal{OBJS}:\ \mathcal{ACT}_{S/A} \neq \emptyset \iff \mathcal{ACT}_{T/A} \neq \emptyset$$

Substituting this into equation 3.6 we see that all active objects in state T were mutually consistent in state S.

$$\begin{aligned} &\forall\ A, B \in \mathcal{OBJS}\ (\mathcal{ACT}_{T/A} \neq \emptyset \ \wedge\ \mathcal{ACT}_{T/B} \neq \emptyset): \\ &\qquad \forall\ x.A \in \mathcal{AS}_{S/A}:\ \forall\ y.B \in \mathcal{R}\ (y.B \prec_{\mathcal{R}} x.A):\ y.B \in \mathcal{AS}_{S/B} \end{aligned} \tag{3.7}$$

From equations 3.4 and 3.5 we see that the states of all active objects remain unchanged between states S and T.

$$\forall\ A \in \mathcal{OBJS}\ (\mathcal{ACT}_{T/A} \neq \emptyset):\ \mathcal{AS}_{T/A}\ =\ \mathcal{AS}_{S/A}$$

Substituting this into equation 3.7 we get the desired result.

$$\begin{aligned} &\forall\ A, B \in \mathcal{OBJS}\ (\mathcal{ACT}_{T/A} \neq \emptyset \ \wedge\ \mathcal{ACT}_{T/B} \neq \emptyset): \\ &\qquad \forall\ x.A \in \mathcal{AS}_{T/A}:\ \forall\ y.B \in \mathcal{R}\ (y.B \prec_{\mathcal{R}} x.A):\ y.B \in \mathcal{AS}_{T/B} \end{aligned} \tag{3.8}$$

That is, the states of all active objects are mutually consistent in state T. $\square$

3.2.2 ACTIVATE Phase Outline

The ACTIVATE phase is responsible for recovering a server's replicas of inactive objects. A server does not begin its ACTIVATE phase until it has completed its JOIN phase. Inactive objects are recovered one at a time and a server coordinates its recovery of an inactive object with those of the other recovering servers of the object (once they have completed their JOIN phases). In order to restore an inactive object, the recovering servers first agree on a new state for the object (one that is consistent with the states of all other active objects in the system) and then alter their logs to reflect this new state. The servers then restore their replicas by replaying the appropriate portions of their logs and begin to receive and process new client requests on the object.

The changes that occur to the system state as a result of the ACTIVATE phase are shown in definition 3.2. Note that the only portion of the state that changes is the portion related to the recovering servers of the inactive object (A).

Definition 3.2

A state, T, solves the ACTIVATE problem for object $A \in \mathcal{OBJS}$ in state S under request structure $(\mathcal{R}, \prec_{\mathcal{R}})$ if T satisfies the following conditions:

AC1. *The new logs of the recovering servers, $(\mathcal{L}_{T/f}, \rightarrow_{T/f})$ $\forall f \in \mathcal{REC}_{S/A}$, are consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$.*

AC2. *The recovering servers of A agree on the object's new state.*

$$\forall f, g \in \mathcal{REC}_{S/A}: \ (\mathcal{L}_{T/f}, \rightarrow_{T/f}) \mid_A \ = \ (\mathcal{L}_{T/g}, \rightarrow_{T/g}) \mid_A$$

AC3. *The new state for object A is consistent with the states of all other active objects.*

$\forall B \in \mathcal{OBJS} \ (\mathcal{ACT}_{T/B} \neq \emptyset):$

$$\forall x.A \in \mathcal{AS}_{T/A}: \ \forall y.B \in \mathcal{R} \ (y.B \prec_{\mathcal{R}} x.A): \ y.B \in \mathcal{AS}_{T/B} \quad \text{and}$$
$$\forall y.B \in \mathcal{AS}_{T/B}: \ \forall x.A \in \mathcal{R} \ (x.A \prec_{\mathcal{R}} y.B): \ x.A \in \mathcal{AS}_{T/A}$$

AC4. *The new logs of the recovering servers preserve the states of any previously active objects.*

$$\forall f \in \mathcal{REC}_{S/A}: \ \forall B \in \mathcal{OBJS}_f \ (f \in \mathcal{ACT}_{S/B}):$$
$$(\mathcal{L}_{T/f}, \rightarrow_{T/f}) \mid_B \ = \ (\mathcal{L}_{S/f}, \rightarrow_{S/f}) \mid_B .$$

AC5. *The only logs affected are those of the recovering servers of A.*

$$\forall f \ \in \ \mathcal{SERV} - \mathcal{REC}_{S/A} \ : \ (\mathcal{L}_{T/f}, \rightarrow_{T/f}) \ = \ (\mathcal{L}_{S/f}, \rightarrow_{S/f})$$

AC6. *The recovering servers of A become active servers of the object.*

$$\mathcal{ACT}_{T/A} = \mathcal{REC}_{S/A} \qquad \forall B \in \mathcal{OBJS} - \{A\}: \ \mathcal{ACT}_{T/B} = \mathcal{ACT}_{S/B}$$
$$\mathcal{REC}_{T/A} = \emptyset \qquad \forall B \in \mathcal{OBJS} - \{A\}: \ \mathcal{REC}_{T/B} = \mathcal{REC}_{S/B}$$

AC7. *The set of failed servers remains the same.*

$$\forall A \in \mathcal{OBJS}: \ \mathcal{FAIL}_{T/A} = \mathcal{FAIL}_{S/A}$$

In addition to meeting these conditions, the recovering servers' new logs should also be as complete as possible, reflecting as many of the previously logged requests as possible. In addition, the new state constructed for object A should be as up to date as possible. The state should reflect all of the logged requests from the time of recovery that are consistent with the current system state. Again, however, we will not formalize these conditions. We present them only as design goals.

The following theorem shows that the ACTIVATE phase preserves consistency within a system.

Theorem 3.2

If S is a state that is observably consistent with a request structure $(\mathcal{R}, \prec_{\mathcal{R}})$, and if T is a state that solves the ACTIVATE problem for object $A \in \mathcal{OBJS}$ in state S, then T is also observably consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$.

Proof: A state is observably consistent with a request structure if it has three properties. First, the logs of all servers in the new state must be consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. From condition AC1 of the ACTIVATE phase definition we know that the logs of all recovering servers of object A, in state T, are consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. From condition AC5 we know that the logs of all other servers remain unchanged from state S, in which they were consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$ by premise. The logs of all servers in state T are therefore consistent with the request structure.

Next, in order for a state to be observably consistent, all active servers of an object must reflect (have logged) the same object state. To see that this property holds in state T, first consider object A. From condition AC6 we know that the only active servers of object A in state T are the servers that were recovering in state S.

$$\mathcal{ACT}_{T/A} = \mathcal{REC}_{S/A}$$

From condition AC2 we know that these servers reflect the same object state for

A in state T.

$$\forall\, f,g \in \mathcal{REC}_{S/A}: \quad (\mathcal{L}_{T/f}, \rightarrow_{T/f})\mid_A \;=\; (\mathcal{L}_{T/g}, \rightarrow_{T/g})\mid_A$$

Now, consider any other active object B ($\mathcal{ACT}_{T/B} \neq \emptyset$) in state T. It follows from condition AC6 that the set of active servers of B remains unchanged between states S and T.

$$\forall\, B \;\in\; \mathcal{OBJS} - \{A\}\;\; (\mathcal{ACT}_{T/B} \neq \emptyset) \;:\; \mathcal{ACT}_{S/B} \;=\; \mathcal{ACT}_{T/B} \tag{3.9}$$

Because S was an observably consistent state, it follows that all of these servers reflected the same object state for B in state S.

$$\forall\, f,g \in \mathcal{ACT}_{T/B}: \quad (\mathcal{L}_{S/f}, \rightarrow_{S/f})\mid_B \;=\; (\mathcal{L}_{S/g}, \rightarrow_{S/g})\mid_B \tag{3.10}$$

From condition AC4 we know that the set of logged requests for object B does not change between states S and T at any of the active servers of B that are recovering servers of A.

$$\forall\, f \in \mathcal{ACT}_{T/B} \bigcap \mathcal{REC}_{S/A}: \quad (\mathcal{L}_{T/f}, \rightarrow_{T/f})\mid_B \;=\; (\mathcal{L}_{S/f}, \rightarrow_{S/f})\mid_B \tag{3.11}$$

From condition AC5 we know that the logs of the other active servers of B (those that are not recovering servers of A) do not change between states S and T and so the set of quests they've logged for B remains the same.

$$\forall\, f \in \mathcal{ACT}_{T/B} - \mathcal{REC}_{S/A}: \quad (\mathcal{L}_{T/f}, \rightarrow_{T/f})\mid_B \;=\; (\mathcal{L}_{S/f}, \rightarrow_{S/f})\mid_B \tag{3.12}$$

Combining equations 3.11 and 3.12 we see that all active servers of B have logged the same set of requests for B in both states S and T.

$$\forall\, f \in \mathcal{ACT}_{T/B}: \quad (\mathcal{L}_{T/f}, \rightarrow_{T/f})\mid_B \;=\; (\mathcal{L}_{S/f}, \rightarrow_{S/f})\mid_B \tag{3.13}$$

Substituting the result of equation 3.13 into equation 3.10 we see that all active servers of B reflect the same object state in state T.

$$\forall\, f,g \in \mathcal{ACT}_{T/B}: \quad (\mathcal{L}_{T/f}, \rightarrow_{T/f})\mid_B \;=\; (\mathcal{L}_{T/g}, \rightarrow_{T/g})\mid_B$$

The last property of observable consistency is that the states of all active objects are mutually consistent. To see that this property holds in the new state, T, consider first any two active objects, $B, C \in \mathcal{OBJS} - \{A\}$, other than A ($\mathcal{ACT}_{T/B} \neq \emptyset$ and $\mathcal{ACT}_{T/C} \neq \emptyset$). From equation 3.9 we know that the set of active servers of these objects does not change between states S and T.

$$\mathcal{ACT}_{S/B} = \mathcal{ACT}_{T/B} \qquad \mathcal{ACT}_{S/C} = \mathcal{ACT}_{T/C}$$

Because S was an observably consistent state, we also know that the active states of these objects were mutually consistent in state S.

$$\begin{aligned} &\forall\, B, C \in \mathcal{OBJS} - \{A\} \;\; (\mathcal{ACT}_{T/B} \neq \emptyset \wedge \mathcal{ACT}_{T/C} \neq \emptyset) : \\ &\qquad \forall\, y.B \in \mathcal{AS}_{S/B} : \; \forall\, z.C \in \mathcal{R}\; (z.C \prec_{\mathcal{R}} y.B) : \; z.C \in \mathcal{AS}_{S/C} \end{aligned} \tag{3.14}$$

From equation 3.13 we know that the states of these active objects do not change between states S and T.

$$\forall\, B \in \mathcal{OBJS} - \{A\} \;\; (\mathcal{ACT}_{T/B} \neq \emptyset) : \; \mathcal{AS}_{T/B} = \mathcal{AS}_{S/B} \tag{3.15}$$

They must therefore remain mutually consistent in state T.

$$\begin{aligned} &\forall\, B, C \in \mathcal{OBJS} - \{A\} \;\; (\mathcal{ACT}_{T/B} \neq \emptyset \wedge \mathcal{ACT}_{T/C} \neq \emptyset) : \\ &\qquad \forall\, y.B \in \mathcal{AS}_{T/B} : \; \forall\, z.C \in \mathcal{R}\; (z.C \prec_{\mathcal{R}} y.B) : \; z.C \in \mathcal{AS}_{T/C} \end{aligned}$$

It follows that any inconsistency between object states in T must involve object A. However, from condition AC3 we know that the active state of A is consistent with the active states of all other objects. The states of all active objects are therefore mutually consistent in state T. $\square$

3.3 Recovery Examples

As an example of the recovery mechanism's behavior, consider again the execution of the resource system shown in figure 3.2. Suppose that server f is the first server to recover after time t_3. At the time server f recovers, the state of the system will be:

$$\mathcal{ACT}_{S/Names} = \emptyset \quad \mathcal{REC}_{S/Names} = \{f\} \quad \mathcal{FAIL}_{S/Names} = \{g\}$$
$$\mathcal{ACT}_{S/Allocations} = \emptyset \quad \mathcal{REC}_{S/Allocations} = \emptyset \quad \mathcal{FAIL}_{S/Allocations} = \{g, h\}$$

$$(\mathcal{L}_{S/f}, \rightarrow_{S/f}): \boxed{reg_1} \quad (\mathcal{L}_{S/g}, \rightarrow_{S/g}): \begin{array}{|c|}\hline reg_2 \\ \hline alc_2 \\ \hline\end{array} \quad (\mathcal{L}_{S/h}, \rightarrow_{S/h}): \boxed{alc_2}$$

Because no objects are active when f recovers, the JOIN phase of f will not take any actions. During its ACTIVATE phase, however, server f will recover its replica of object "Names". Because no objects are active, server f is free to recover any valid state of "Names" for its replica; it does not have to be concerned with ensuring consistency with the states of any other active objects. Server f therefore recovers its replica using the state reflected in its log (the state reflecting only the registration of client 1). The resulting state is shown below:

$$\mathcal{ACT}_{S/Names} = \{f\} \quad \mathcal{REC}_{S/Names} = \emptyset \quad \mathcal{FAIL}_{S/Names} = \{g\}$$
$$\mathcal{ACT}_{S/Allocations} = \emptyset \quad \mathcal{REC}_{S/Allocations} = \emptyset \quad \mathcal{FAIL}_{S/Allocations} = \{g, h\}$$

$$(\mathcal{L}_{S/f}, \rightarrow_{S/f}): \boxed{reg_1} \quad (\mathcal{L}_{S/g}, \rightarrow_{S/g}): \begin{array}{|c|}\hline reg_2 \\ \hline alc_2 \\ \hline\end{array} \quad (\mathcal{L}_{S/h}, \rightarrow_{S/h}): \boxed{alc_2}$$

Now, suppose that server h is the next server to recover. Again, no objects served by h are active at the time of the recovery and so the server's JOIN phase will not take any actions. Instead, server h's replica of "Allocations" is recovered during its ACTIVATE phase. Unlike the recovery of object "Names" by server f, however, server g is not free to recover any state for object "Allocations"; it must ensure that the state recovered is one that is consistent with the state of the now active object "Names". Server h must therefore delete request alc_2 from its log because the registration of client 2 is not reflected in the active state of the system. The state of the system resulting from the recovery of h will then be:

$$\mathcal{ACT}_{S/Names} = \{f\} \quad \mathcal{REC}_{S/Names} = \emptyset \quad \mathcal{FAIL}_{S/Names} = \{g\}$$
$$\mathcal{ACT}_{S/Allocations} = \{h\} \quad \mathcal{REC}_{S/Allocations} = \emptyset \quad \mathcal{FAIL}_{S/Allocations} = \{g\}$$

$$(\mathcal{L}_{S/f}, \rightarrow_{S/f}): \boxed{reg_1} \quad (\mathcal{L}_{S/g}, \rightarrow_{S/g}): \begin{array}{|c|}\hline reg_2 \\ \hline alc_2 \\ \hline\end{array} \quad (\mathcal{L}_{S/h}, \rightarrow_{S/h}): \boxed{\emptyset}$$

If server g then recovers last, both objects it servers will be active. The states of these objects are therefore transferred to g during its JOIN phase and placed in its log. No actions are taken during g's ACTIVATE phase. The final state of the system (after the recovery of all three servers) is shown below:

$$\mathcal{ACT}_{S/Names} = \{f, g\} \quad \mathcal{REC}_{S/Names} = \emptyset \quad \mathcal{FAIL}_{S/Names} = \emptyset$$
$$\mathcal{ACT}_{S/Allocations} = \{g, h\} \quad \mathcal{REC}_{S/Allocations} = \emptyset \quad \mathcal{FAIL}_{S/Allocations} = \emptyset$$

$$(\mathcal{L}_{S/f}, \rightarrow_{S/f}): \boxed{reg_1} \quad (\mathcal{L}_{S/g}, \rightarrow_{S/g}): \boxed{reg_1} \quad (\mathcal{L}_{S/h}, \rightarrow_{S/h}): \boxed{\emptyset}$$

As another example, suppose that server f recovers first as above, but that servers g and h then recover simultaneously. Again, the JOIN phase of h will not take any actions because the object served by h ("Allocations") is inactive at the time of the recovery. Because object "Names" is active, though, the JOIN phase of g will recover g's replica of that object. In order to restore the replica to the object's current active state, the JOIN phase of g adds request reg_1 to g's log and deletes request reg_2. Note, however, that in order to preserve consistency within the log of g, request alc_2 must also be deleted because it depends on request reg_2. The state of the system immediately after the JOIN phases of servers g and h will then be:

$$\mathcal{ACT}_{S/Names} = \{f, g\} \quad \mathcal{REC}_{S/Names} = \emptyset \quad \mathcal{FAIL}_{S/Names} = \emptyset$$
$$\mathcal{ACT}_{S/Allocations} = \emptyset \quad \mathcal{REC}_{S/Allocations} = \{g, h\} \quad \mathcal{FAIL}_{S/Allocations} = \emptyset$$

$$(\mathcal{L}_{S/f}, \rightarrow_{S/f}): \boxed{reg_1} \quad (\mathcal{L}_{S/g}, \rightarrow_{S/g}): \boxed{reg_1} \quad (\mathcal{L}_{S/h}, \rightarrow_{S/h}): \boxed{alc_2}$$

After completing their JOIN phases, servers g and h begin their ACTIVATE phases. During their ACTIVATE phases, the servers recover their replicas of object "Allocations". The servers cooperate in deciding on a new state for the object. Because the only request on the object known to either server (alc_2) is inconsistent with the active state of "Names", the servers will decide on a state that reflects no allocation of resources. The final system state is the same as that in the previous example.

As a final example, suppose that server h is the first server to recover. No objects will be active at the time of the recovery, so no actions will be taken during the JOIN phase of h. During its ACTIVATE phase, though, server h will recover its replica of "Allocations" in the state reflected by its log (the state reflecting the allocation made to client 2).

Suppose now that servers f and g simultaneously recover. The state of the system at the time of the servers recovery will then be:

$$\mathcal{ACT}_{S/Names} = \emptyset \quad \mathcal{REC}_{S/Names} = \{f, g\} \quad \mathcal{FAIL}_{S/Names} = \emptyset$$
$$\mathcal{ACT}_{S/Allocations} = \{h\} \quad \mathcal{REC}_{S/Allocations} = \{g\} \quad \mathcal{FAIL}_{S/Allocations} = \emptyset$$

$$(\mathcal{L}_{S/f}, \rightarrow_{S/f}): \boxed{reg_1} \quad (\mathcal{L}_{S/g}, \rightarrow_{S/g}): \boxed{\begin{array}{c} reg_2 \\ \hline alc_2 \end{array}} \quad (\mathcal{L}_{S/h}, \rightarrow_{S/h}): \boxed{alc_2}$$

During its JOIN phase, server g will recover its replica of "Allocations". Because its log already reflects the current state of that object, no alterations are made to the log. No actions are taken during the JOIN phase of server f.

When servers f and g enter their ACTIVATE phases, they recover their replicas of object "Names". The servers merge their logs to form a new state for the object that reflects both the registrations of client 1 and client 2. Server f alters its log to reflect this new state by adding in request reg_2. Server g similarly alters its log by adding in request reg_1. The resulting system state is then:

$$\mathcal{ACT}_{S/Names} = \{f, g\} \quad \mathcal{REC}_{S/Names} = \emptyset \quad \mathcal{FAIL}_{S/Names} = \emptyset$$
$$\mathcal{ACT}_{S/Allocations} = \{g, h\} \quad \mathcal{REC}_{S/Allocations} = \emptyset \quad \mathcal{FAIL}_{S/Allocations} = \emptyset$$

$$(\mathcal{L}_{S/f}, \rightarrow_{S/f}): \begin{array}{|c|}\hline reg_1 \\ \hline reg_2 \\ \hline \end{array} \quad (\mathcal{L}_{S/g}, \rightarrow_{S/g}): \begin{array}{|c|}\hline reg_2 \\ \hline reg_1 \\ \hline alc_2 \\ \hline \end{array} \quad (\mathcal{L}_{S/h}, \rightarrow_{S/h}): \begin{array}{|c|}\hline alc_2 \\ \hline \end{array}$$

Note that request reg_2 must be included in the new state of "Names" because the active state of "Allocations" depends on it.

3.4 Summary

In this chapter we examined the problem of how inconsistencies arise between the states of objects in a system. Inconsistencies can develop in two ways. First, inconsistencies develop between replicas of the same object when recovering servers fail to restore the states of their replicas to those held by other servers in the system. Second, inconsistencies can occur between the states of different objects when recovering servers restore old and out of date object states.

A recovery algorithm was outlined for preventing these inconsistencies when a server fails. The algorithm was divided into two phases based on the two types inconsistencies that occur between objects and replicas.

JOIN phase	Restore a server's replicas of active objects to the current active states of those objects.
ACTIVATE phase	Restore a server's replicas of inactive objects to states that are consistent with the states of all active objects in the system. This phase had the additional property that all recovering

servers of an inactive object agreed on the state restored for that object.

The behaviors of the recovery phases were formally described and it was proved that these behaviors preserve consistency within a system.

The chapter concluded with several examples of how the recovery mechanism restores consistent states to servers' object replicas.

Chapter 4

Log Transformations

The main difficulty involved in implementing the recovery phases of the previous chapter is ensuring that the alterations that occur to servers' logs preserve the consistency of those logs. This chapter presents functions for adding and deleting requests from a server's log in a way that preserves the log's consistency. These functions (or transformations) will form the basis of our recovery algorithms.

4.1 Log Addition

In order to bring a recovering server's log into a state that is consistent with the rest of the system, it is sometimes necessary to add requests to the log. Such added requests are generally requests that the server missed receiving because of its failure. For example, consider the execution shown in figure 4.1. In this execution, servers f and g fail after receiving the registration of client 1 but before receiving the registration of client 2. Server h remains active throughout the execution and receives the allocation request (alc_2) from client 2. This request is not received by server g, however, because g fails before its delivery. If server g recovers at time t_2, it will have to add this request to its log so that the log reflects the current state of "Allocations" (*i.e.* the state reflected in the log of server h).

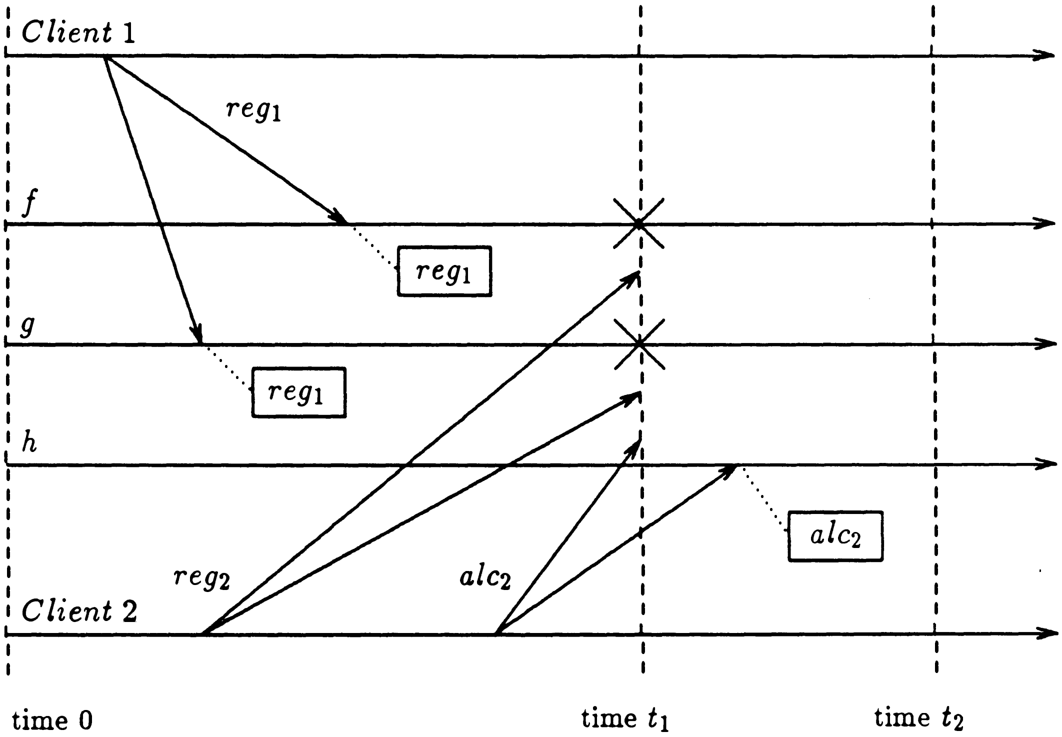


Figure 4.1: A recovery requiring addition to a log

$$\text{add}_Q(\mathcal{L}_f, \rightarrow_f) \;=\; (\mathcal{L}, \rightarrow_\mathcal{L})$$

where

$$\mathcal{L} \;=\; \mathcal{L}_f \bigcup Q \bigcup \left[\bigcup_{x.A \in Q} \; \bigcup_{B \in \mathcal{OBJS}_f} \mathcal{DEP}_B(x.A) \right]$$

$\rightarrow_\mathcal{L}$ is any extension of $\rightarrow_f$ consistent with $\prec_R$.

Figure 4.2: Log addition preserving consistency

The addition of requests to a server's log can cause the log to become inconsistent, however. In the above example, the log of server g becomes inconsistent when request alc_2 is added because the client registration on which alc_2 depends (reg_2) is missing from the log. In order to preserve consistency within a log, any dependents of an added request must also be added to the log (unless they are already present).

Definition 4.1

The set of object B dependents of request $x.A$ are

$$\mathcal{DEP}_B(x.A) \;=\; \{y.B \in \mathcal{R} \mid y.B \prec_\mathcal{R} x.A\}$$

Shown below is the complete sequence of changes required to consistently add request alc_2 to the log of server g:

$$\begin{array}{|c|}\hline reg_1 \\ \hline\end{array} \;\Longrightarrow\; \begin{array}{|c|}\hline reg_1 \\ \hline alc_2 \\ \hline\end{array} \;\Longrightarrow\; \begin{array}{|c|}\hline reg_1 \\ \hline reg_2 \\ \hline alc_2 \\ \hline\end{array}$$

Figure 4.2 presents a function for adding a set of requests, $Q \subset \mathcal{R}$, to the log of a server, $f \in \mathcal{SERV}$. As shown in the following theorem, this function preserves the consistency of the log.

Theorem 4.1

If $(\mathcal{L}_f, \rightarrow_f)$ is a log for server f consistent with request structure $(\mathcal{R}, \prec_{\mathcal{R}})$, and if $Q \subset \mathcal{R}$ is a set of requests on objects served by f, then $\mathbf{add}_Q(\mathcal{L}_f, \rightarrow_f)$ is also consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$.

Proof: Let $(\mathcal{L}, \rightarrow_{\mathcal{L}}) = \mathbf{add}_Q(\mathcal{L}_f, \rightarrow_f)$. We first show that $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ only contains requests on objects served by f. By premise, $(\mathcal{L}_f, \rightarrow_f)$ is consistent and so only contains requests on objects served by f. The only requests added to this log by the function are those in Q and its dependents. By premise, all of the requests in Q are on objects served by f. From the definition of the log addition function, the only dependent requests added to the log are those on objects served by f. All of the requests added to the log are therefore on objects served by f.

We now show that, for any request in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$, all of its dependents (on objects served by f) are also in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$. Let $x.A \in \mathcal{L}$ be any request in the new log. There are three cases:

<u>Case 1:</u> $x.A \in \mathcal{L}_f$

By premise, $(\mathcal{L}_f, \rightarrow_f)$ is consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$ and so all dependents of $x.A$ (on objects served by f) are in $\mathcal{L}_f$. Because $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ is formed by adding requests to $(\mathcal{L}_f, \rightarrow_f)$, it follows that these dependents remain in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$.

<u>Case 2:</u> $x.A \in Q$

It follows immediately from the definition of the log addition function that all of the dependents of $x.A$ (on objects served by f) are added to $(\mathcal{L}, \rightarrow_{\mathcal{L}})$.

<u>Case 3:</u> $x.A \notin \mathcal{L}_f \land x.A \notin Q$

Request $x.A$ must have been added to $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ because it is a dependent of some request, $y.B$, in Q.

$$x.A \prec_{\mathcal{R}} y.B \tag{4.1}$$

Let $z.C \in \mathcal{R}$ be any dependent of request $x.A$ made on an object served by f $(C \in \mathcal{OBJS}_f)$.

$$z.C \prec_{\mathcal{R}} x.A \tag{4.2}$$

Because $\prec_{\mathcal{R}}$ is transitive, it follows from equations 4.1 and 4.2 that request $y.B$ is also dependent on $z.C$.

$$z.C \prec_{\mathcal{R}} y.B$$

From the definition of the log addition function it follows immediately then that request $z.C$ is added to $(\mathcal{L}, \rightarrow_{\mathcal{L}})$.

The last thing we must show is that the order of requests in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ is consistent with $\prec_{\mathcal{R}}$. However, this follows immediately from the definition of the log addition function. □

4.2 Log Deletion

In addition to adding requests to its log, a recovering server may also need to delete requests from its log in order to bring it into consistency with the rest of the system. Such deleted requests are generally requests that were not recovered as part of their object's states by previously recovering servers of the objects. For example, consider the execution shown in figure 4.3. Suppose server f recovers first and restores its replica of "Names" from its log. The state of "Names" will then only reflect the registration of client 1 (reg_1); it will not reflect the registration of client 2 (reg_2). If server g recovers next, it will have to delete request reg_2 from its log in order to bring it into consistency with f.

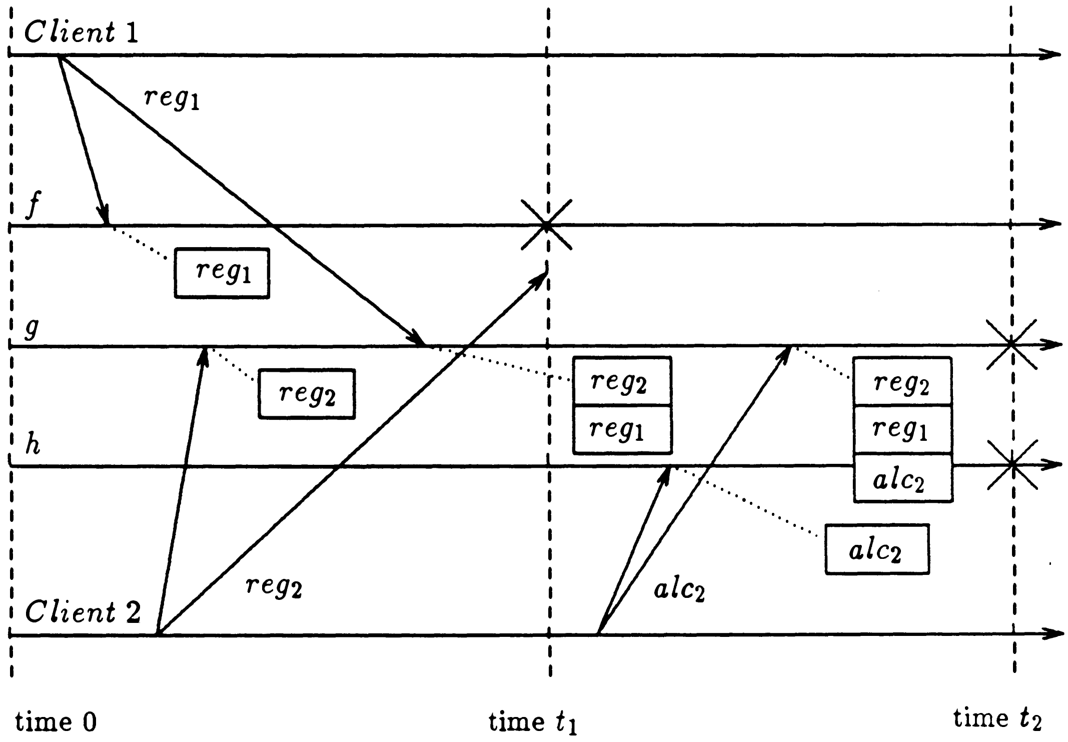


Figure 4.3: A recovery requiring deletion from a log

$$\mathbf{delete}_Q(\mathcal{L}_f, \rightarrow_f) \;=\; (\mathcal{L}, \rightarrow_{\mathcal{L}})$$

$$\text{where}$$

$$\mathcal{L} \;=\; \{\; x.A \in \mathcal{L}_f \mid x.A \notin Q \;\wedge\; \nexists\, y.B \in Q :\; y.B \prec_{\mathcal{R}} x.A \;\}$$

$$\forall\, x.A, y.B \in \mathcal{L} :\; (x.A \rightarrow_{\mathcal{L}} y.B) \Leftrightarrow (x.A \rightarrow_f y.B)$$

Figure 4.4: Log deletion preserving consistency

Like the addition of requests, the deletion of requests can cause a server's log to become inconsistent. In the previous example, the log of server g becomes inconsistent when request reg_2 is deleted because the allocation that depends on it (alc_2) is still present in the log. In order to preserve consistency within a log, any requests that depend on a deleted request must also be removed from the log. Illustrated below is the complete sequence of changes required to remove request reg_2 from the log of server g:

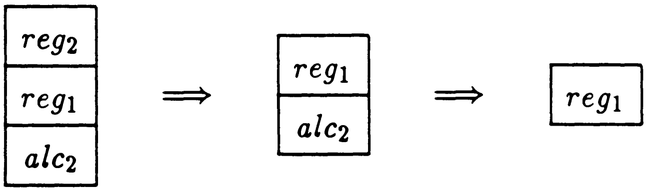


Figure 4.4 presents a function for deleting a set of requests, Q, from the log of a server, f. As shown in the following theorem, this function preserves the consistency of the log.

Theorem 4.2

If $(\mathcal{L}_f, \rightarrow_f)$ is a log for server f consistent with request structure $(\mathcal{R}, \prec_{\mathcal{R}})$, and if $Q \subset \mathcal{L}_f$ is a subset of the requests in $(\mathcal{L}_f, \rightarrow_f)$, then $\mathbf{delete}_Q(\mathcal{L}_f, \rightarrow_f)$ *is also consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$.*

Proof: Let $(\mathcal{L}, \rightarrow_{\mathcal{L}}) = \mathbf{delete}_Q(\mathcal{L}_f, \rightarrow_f)$. We first show that $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ only contains requests on objects served by f. From the definition of the log deletion function, the requests in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ are a subset of the requests in $(\mathcal{L}_f, \rightarrow_f)$. By premise, $(\mathcal{L}_f, \rightarrow_f)$ is consistent and so these requests must all be on objects served by f.

We now show that, for any request in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$, all of its dependents (on objects served by f) are also in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$. The proof is by contradiction. Let $x.A$ be any request in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$. Suppose some dependent of $x.A$ (made on an object served by f) is missing from $(\mathcal{L}, \rightarrow_{\mathcal{L}})$. Let $y.B$ denote this dependent.

$$y.B \prec_{\mathcal{R}} x.A \tag{4.3}$$

From above, we know that $\mathcal{L} \subseteq \mathcal{L}_f$ and so request $x.A$ is in $(\mathcal{L}_f, \rightarrow_f)$. Because $(\mathcal{L}_f, \rightarrow_f)$ is consistent, it follows that request $y.B$ is also in $(\mathcal{L}_f, \rightarrow_f)$. Request $y.B$ must therefore have been removed from the log by the log deletion function when forming $(\mathcal{L}, \rightarrow_{\mathcal{L}})$. This could have happened for one of two reasons: either it was in Q or it was dependent on a request in Q.

If request $y.B$ were in Q, then request $x.A$ would also have been removed from the log by the transformation because it depends on $y.B$ (a request in Q), a contradiction. Request $y.B$ must therefore have been removed from the log because it depends on some request, $z.C$, in Q.

$$z.C \prec_{\mathcal{R}} y.B \tag{4.4}$$

Because $\prec_{\mathcal{R}}$ is transitive, it follows from equations 4.3 and 4.4 that request $x.A$ is also dependent on $z.C$.

$$z.C \prec_{\mathcal{R}} x.A$$

Request $x.A$ should therefore have been removed from the log because it depends on a request in Q, another contradiction. The new log, $(\mathcal{L}, \rightarrow_{\mathcal{L}})$, must therefore contain $y.B$.

The last thing we must show is that the order of requests in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ is consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. From the definition of the log deletion function, the requests in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ are ordered the same way they were in $(\mathcal{L}_f, \rightarrow_f)$. Because $(\mathcal{L}_f, \rightarrow_f)$ is consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$, it follows that this order is consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. $\square$

4.3 Using Dependency Estimates

The previous log transformations were both based on having explicit knowledge of the dependencies between requests. Such information is not available in all systems, however. When the exact set of clients is either unknown to the servers, or is large and dynamically changing, it can be difficult or impossible to maintain explicit dependency information. When this information is not available to the servers, the preceding transformations cannot be used.

This section examines how the log transformations can be modified to use estimates of the true dependencies. The key to the success of these new transformations will be the use of estimates that never under-estimate the true set of the dependencies in the system. We refer to estimate that have this property as *sound estimates*. By using sound estimates, the transformations will enforce some extraneous orderings because of the inaccuracy of the estimates, but they will also enforce all true dependencies. The actual estimates used in the new transformations are presented later in chapter 6.

4.3.1 Log Addition

Consider first the problem of adding a set of requests to a server's log. Let $\overline{\mathcal{DEP}}_B(x.A)$ denote any sound estimate of the set of object B dependents of request $x.A$.

$$\mathcal{DEP}_B(x.A) \subseteq \overline{\mathcal{DEP}}_B(x.A) \tag{4.5}$$

We would like to modify the log addition transformation, $\mathbf{add}_Q(\mathcal{L}_f, \rightarrow_f)$, to use $\overline{\mathcal{DEP}}_B(x.A)$ instead of the true dependency set $\mathcal{DEP}_B(x.A)$. Unfortunately, as we show below, the estimate cannot be used directly in place of $\mathcal{DEP}_B(x.A)$. The reason for this is that the log addition transformation uses the transitive property of causal dependencies in order to preserve consistency within a server's log.

$$z.C \prec_{\mathcal{R}} y.B \bigwedge y.B \prec_{\mathcal{R}} x.A \quad \Longrightarrow \quad z.C \prec_{\mathcal{R}} x.A$$

The estimate does not have this transitive property.

$$z.C \in \overline{\mathcal{DEP}}_C(y.B) \bigwedge y.B \in \overline{\mathcal{DEP}}_B(x.A) \quad \not\Rightarrow \quad z.C \in \overline{\mathcal{DEP}}_C(x.A)$$

It may seem counter-intuitive that an estimate would not have the transitive property. However, in the estimates we describe later, an estimate may be able to find evidence contradicting a dependency such as $z.C \rightarrow x.A$ without finding evidence to contradict either of the dependencies $z.C \rightarrow y.B$ or $y.B \rightarrow x.A$. The estimate can then determine that it is not the case that both $z.C \rightarrow y.B$ and $y.B \rightarrow x.A$ hold. But, it cannot determine which one, if any, is the real dependency.

To illustrate how this creates problems in the log addition transformation, consider the transformation $\mathbf{add}_Q(\mathcal{L}_f, \rightarrow_f)$. Let $x.A$ be any of the requests in Q added to $(\mathcal{L}_f, \rightarrow_f)$. In order to preserve consistency in the log, the addition transformation explicitly adds each dependent of $x.A$ to the log. For each of these dependents, $y.B$, the addition transformation also automatically adds each of its dependents to the log because, by the transitivity of the request dependency relation, each of these dependents is also a dependent of $x.A$. Thus, for each request added to the log, all of its dependents are also assured of being added to the log.

However, if an estimate is used, some dependents of added requests may be omitted from the log. If request $y.B$ is added to the log because it is an estimated dependent of $x.A$ (it might not be a real dependent), then the transformation should also add to the log all estimated dependents of $y.B$, in order to preserve the consistency of the log. From the definition of the transformation, though, only estimated dependents of $x.A$ would be added to the log. It is possible that

1. $R = 0$
2. $\mathcal{L}^{(0)} = \mathcal{L}_f \cup Q$
3. $NEWREQS^{(0)} = \mathcal{L}^{(0)} - \mathcal{L}_f$
4. **while** $NEWREQS^{(R)} \neq \emptyset$

4.1 $R = R + 1$

4.2 $\mathcal{L}^{(R)} = \mathcal{L}^{(R-1)} \cup \left[\bigcup_{B \in \mathcal{OBJS}_f} \; \bigcup_{x.A \in NEWREQS^{(R-1)}} \overline{\mathcal{DEP}}_B(x.A) \right]$

4.3 $NEWREQS^{(R)} = \mathcal{L}^{(R)} - \mathcal{L}^{(R-1)}$

Figure 4.5: Iterative addition of requests

some of the estimated dependents of $y.B$ may not be estimated dependents of $x.A$. These extra estimated dependents would be omitted from the log, creating an inconsistency.

In order to use the dependency set estimate, the log addition transformation, must add requests to a log iteratively. In each round of the iteration, the transformation adds to the log the estimated dependents of the requests added in the previous round. An algorithm for determining the complete set of requests in the transformed log using this addition scheme is shown in figure 4.5. In the algorithm, R is the round number, $NEWREQS^{(R)}$ is the set of new requests added to the log in round R, and $\mathcal{L}^{(R)}$ is the complete set of requests contained in the log after round R.

The complete log addition transformation using this algorithm is presented in figure 4.6. As shown in the following theorem, this transformation preserves the consistency of a log.

$$\textbf{add}_Q(\mathcal{L}_f, \rightarrow_f) \;=\; (\mathcal{L}, \rightarrow_{\mathcal{L}})$$

where

$$\mathcal{L} \;=\; \mathcal{L}^{(R^*)} \qquad\qquad R^* \;=\; \text{MIN}\{\; R \mid \mathcal{L}^{(R)} = \mathcal{L}^{(R+1)} \;\}$$

$\rightarrow_{\mathcal{L}}$ is any extension of $\rightarrow_f$ consistent with $\overline{\mathcal{DEP}}_B(x.A)$.

Figure 4.6: Log addition using estimates

Theorem 4.3

If $(\mathcal{L}_f, \rightarrow_f)$ is a log for server f consistent with request structure $(\mathcal{R}, \prec_{\mathcal{R}})$, and if $Q \subseteq \mathcal{R}$ is a set of requests on objects served by f, then $\textbf{add}_Q(\mathcal{L}_f, \rightarrow_f)$ is also consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$.

Proof: Let $(\mathcal{L}, \rightarrow_{\mathcal{L}}) = \textbf{add}_Q(\mathcal{L}_f, \rightarrow_f)$. We first show that $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ only contains requests on objects served by f. By premise, both $(\mathcal{L}_f, \rightarrow_f)$ and Q only contain requests on objects served by f. It thus follows immediately that $\mathcal{L}^{(0)}$ only contains requests on objects served by f. In each round of the addition iteration, only requests on objects served by f are added to the log. It therefore follows by induction that each $\mathcal{L}^{(R)}$ only contains requests on objects served by f.

We now show that, for any request in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$, all of its dependents (on objects served by f) are also in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$. Let $x.A \in \mathcal{L}$ be any request in the new log. There are two cases:

Case 1: $x.A \in \mathcal{L}_f$

By premise, $(\mathcal{L}_f, \rightarrow_f)$ is consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$ and so all of the dependents of $x.A$ (on objects served by f) are in $\mathcal{L}_f$. Because $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ is formed by adding requests to log $(\mathcal{L}_f, \rightarrow_f)$, it follows that the dependents remain in

$(\mathcal{L}, \rightarrow_{\mathcal{L}})$.

Case 2: $x.A \in NEWREQS^{(R)}$ (*i.e.* $x.A$ was added in round R)

From the definition of the iterative addition algorithm, all of the dependents of $x.A$ (on objects served by f) are added to the log in round $R+1$.

The last thing we must show is that the order of requests in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ is consistent with $\prec_{\mathcal{R}}$. By definition, $\rightarrow_{\mathcal{L}}$ is consistent with $\overline{\mathcal{DEP}}_B(x.A)$. From property 4.5 of the estimate, it follows that if two requests, $x.A, y.B \in \mathcal{L}$, are related ($y.B \prec_{\mathcal{R}} x.A$) then $y.B \in \overline{\mathcal{DEP}}_B(x.A)$ and so these requests are properly ordered in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$. □

4.3.2 Log Deletion

Consider now the problem of deleting a set of requests from a log. We would like to modify the log deletion transformation to use an estimate of the relationship between requests. Let $\overline{\mathcal{CON}}(x.A \prec y.B)$ denote any such sound estimate.

$$\forall\ x.A, y.B \in \mathcal{R} : \overline{\mathcal{CON}}(x.A \prec y.B) \implies x.A \not\prec_{\mathcal{R}} y.B \tag{4.6}$$

Note that $\overline{\mathcal{CON}}(x.A \prec y.B)$ estimates the predicate that two requests are *unrelated.*

As with the log addition transformation, this estimate cannot be used directly in the log deletion transformation. If it were, inconsistencies could occur in the transformed logs because the transformation may fail to remove all requests that depend on the deleted requests. In order to use the estimate, the log deletion transformation must iteratively delete requests from a log. An algorithm for doing this is shown in figure 4.7. In the algorithm R is the round number, $DELETED^{(R)}$ is the set of requests deleted from the log in round R, and $\mathcal{L}^{(R)}$ is the set of requests contained in the log after round R.

1. $R \; = \; 0$

2. $\mathcal{L}^{(0)} \; = \; \mathcal{L}_f - Q$

3. $DELETED^{(0)} \; = \; \mathcal{L}_f - \mathcal{L}^{(0)}$

4. **while** $DELETED^{(R)} \neq \emptyset$

4.1 $\quad R \; = \; R+1$

4.2 $\quad \mathcal{L}^{(R)} \; = \; \{ \; y.B \in \mathcal{L}^{(R-1)} \mid$
$\qquad\qquad \forall \; x.A \in DELETED^{(R-1)} : \; \overline{\mathcal{CON}}(x.A \prec y.B) \; \}$

4.3 $\quad DELETED^{(R)} \; = \; \mathcal{L}^{(R-1)} - \mathcal{L}^{(R)}$

Figure 4.7: Iterative deletion of requests

$$\mathbf{delete}_Q(\mathcal{L}_f, \rightarrow_f) \;\; = \;\; (\mathcal{L}, \rightarrow_{\mathcal{L}})$$

where

$$\mathcal{L} \; = \; \mathcal{L}^{(R^*)} \qquad\qquad R^* \; = \; \mathrm{MIN}\{ \; R \mid \mathcal{L}^{(R)} = \mathcal{L}^{(R+1)} \; \}$$

$$\forall \; x.A, y.B \in \mathcal{L} : \; (x.A \rightarrow_{\mathcal{L}} y.B) \Leftrightarrow (x.A \rightarrow_f y.B)$$

Figure 4.8: Log deletion using estimates

The complete log deletion transformation using this algorithm is presented in figure 4.8. As shown in the following theorem, this transformation preserves the consistency of a log.

Theorem 4.4

If $(\mathcal{L}_f, \rightarrow_f)$ *is a log for server* f *consistent with request structure* $(\mathcal{R}, \prec_{\mathcal{R}})$, *and if* $Q \subseteq \mathcal{L}_f$ *is a subset of the requests in* $(\mathcal{L}_f, \rightarrow_f)$, *then* $\mathbf{delete}_Q(\mathcal{L}_f, \rightarrow_f)$ *is also consistent with* $(\mathcal{R}, \prec_{\mathcal{R}})$.

Proof: Let $(\mathcal{L}, \rightarrow_{\mathcal{L}}) = \mathbf{delete}_Q(\mathcal{L}_f, \rightarrow_f)$. We first show that $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ only contains requests on objects served by f. Because $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ is formed by deleting requests from $(\mathcal{L}_f, \rightarrow_f)$, we know that $\mathcal{L} \subseteq \mathcal{L}_f$. By premise, $(\mathcal{L}_f, \rightarrow_f)$ is consistent and so all of these requests are on objects served by f.

We now show that, for any request in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$, all of its dependents (on objects served by f) are also in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$. The proof is by contradiction. Let $x.A \in \mathcal{L}$ be any request in the transformed log, and let $y.B \in \mathcal{R}$ be any of its dependents ($y.B \prec_{\mathcal{R}} x.A$) on an object served by f ($B \in \mathcal{OBJS}_f$). Suppose that $y.B$ is not in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$. Because $\mathcal{L} \subseteq \mathcal{L}_f$ we know that $x.A \in \mathcal{L}_f$. Because $(\mathcal{L}_f, \rightarrow_f)$ is consistent by premise, it follows that $y.B \in \mathcal{L}_f$. Request $y.B$ must therefore have been removed from the log in some round, R, of the iterative deletion algorithm. However, by definition of the algorithm, request $x.A$ would then have been removed from the log in round $R+1$ of the iteration because it depends on request $y.B$, contradicting the fact that $x.A \in \mathcal{L}$. The transformed log, $(\mathcal{L}, \rightarrow_{\mathcal{L}})$, must therefore contain $y.B$.

The last thing we must show is that the order of requests in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ is consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. However, by definition, the order of requests in $(\mathcal{L}, \rightarrow_{\mathcal{L}})$ is consistent with the order of requests in $(\mathcal{L}_f, \rightarrow_f)$, which is by premise consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. $\square$

4.4 Summary

This chapter presented several transformations for altering the log of a server while preserving its consistency. The chapter began by presenting transformations for adding and deleting requests from a log. These transformations were based on having explicit knowledge of the dependencies between client requests. It was then shown how these transformations can be modified to use estimates of the request dependencies when exact information is not available. A key to the correctness of these new transformations was the use of approximations that never under-estimated the true set of dependencies. By using sound estimates, the transformations were assured of enforcing all true dependencies, in addition to a few extraneous ones.

Chapter 5

Recovery Solutions

In this chapter we present algorithms for solving the JOIN and ACTIVATE problems. These algorithms are based on the log transformations of chapter 4. We begin by assuming that explicit dependency information is not available in the system and so the only transformations available to the recovery algorithms are those based on dependency estimates. We then show how these recovery algorithms can be simplified when the transformations using explicit dependency information are available.

5.1 JOIN Solution

When a server first recovers from a failure it restores its replicas of active objects to those objects' current states. The server alters its log to reflect the current object states and then replays the log to restore its replicas.

A recovering server's log may be out of date with respect to the current states of active objects in two ways. First, the log may not reflect all of the requests present in those objects' current states. Such requests are generally those that the server did not received while it was failed. We let $\mathcal{MS}_{S/f}$ denote the set of requests on active objects missing from the log of a recovering server, f, in state

S.

$$\mathcal{MS}_{S/f} = \bigcup_{\{ A \in \mathcal{OBJS}_f \mid \mathcal{ACT}_{S/A} \neq \emptyset \}} [\mathcal{AS}_{S/A} - (\mathcal{L}_{S/f}, \rightarrow_{S/f}) |_A]$$

Second, a recovering server's log may be out of date because it reflects requests on active objects that are not present in those objects' current states. Such requests are generally those that the active servers failed to recover after some previous failure event. We let $\mathcal{NR}_{S/f}$ denote the set of requests on active objects present in the log of server f, in state S, that are not present in their objects' active states.

$$\mathcal{NR}_{S/f} = \bigcup_{\{ A \in \mathcal{OBJS}_f \mid \mathcal{ACT}_{S/A} \neq \emptyset \}} [(\mathcal{L}_{S/f}, \rightarrow_{S/f}) |_A - \mathcal{AS}_{S/A}]$$

In order to restore correct object replicas, a recovering server must remove the requests in $\mathcal{NR}_{S/f}$ from its log and add those in $\mathcal{MS}_{S/f}$. The complete algorithm for solving the JOIN problem for server f in state S is shown in figure 5.1. In the algorithm, T is the state constructed to solve the problem.

Note that in step JS1 the new log is tested to make sure that the addition and deletion of requests yielded the correct logged state. The reason for this is that the transformations may inadvertently attempt to add or delete a request from the active state logged for an object. Because dependency estimates are used, the log transformations may occasionally incorrectly believe that a dependency holds between two requests, one of which is in its object's active state and the other of which is not. When this happens, the transformations may incorrectly add or delete requests from the logged state of an active object in order to preserve the log's consistency. When this situation occurs, the recovery algorithm must abort and wait until better estimates of the dependencies can be formed. The technique of *recovery logs* [Gra78] (do not confuse this with the term "log" used in this dissertation) can be used to record and undo any changes to a server's log resulting from an aborted recovery attempt.

The JOIN recovery algorithm is formally proved correct below:

JS1. $(\mathcal{L}_{T/f}, \rightarrow_{T/f}) \; = \; \mathbf{add}_{\mathcal{MS}_{S/f}}(\mathbf{delete}_{\mathcal{NR}_{S/f}}(\mathcal{L}_{S/f}, \rightarrow_{S/f}))$

if $\exists\, A \in \mathcal{OBJS}_f \; (\mathcal{ACT}_{S/A} \neq \emptyset)$ *s.t.* $(\mathcal{L}_{T/f}, \rightarrow_{T/f}) \,|_A \; \neq \; \mathcal{AS}_{S/A}$

then abort

JS2. $(\mathcal{L}_{T/g}, \rightarrow_{T/g}) \; = \; (\mathcal{L}_{S/g}, \rightarrow_{S/g}) \qquad \forall\, g \; \in \; \mathcal{SERV} - \{f\}$

JS3. $$\left.\begin{aligned} \mathcal{ACT}_{T/A} &= \mathcal{ACT}_{S/A} \cup \{f\} \\ \mathcal{REC}_{T/A} &= \mathcal{REC}_{S/A} - \{f\} \end{aligned}\right. \qquad \forall\, A \in \mathcal{OBJS}_f \; (\mathcal{ACT}_{S/A} \neq \emptyset)$$

$$\left.\begin{aligned} \mathcal{ACT}_{T/A} &= \mathcal{ACT}_{S/A} \\ \mathcal{REC}_{T/A} &= \mathcal{REC}_{S/A} \end{aligned}\right. \qquad \forall\, A \in \mathcal{OBJS} \; (A \notin \mathcal{OBJS}_f \; \vee \mathcal{ACT}_{S/A} = \emptyset)$$

JS4. $\mathcal{FAIL}_{T/A} \; = \; \mathcal{FAIL}_{S/A} \qquad \forall\, A \in \mathcal{OBJS}$

Figure 5.1: Solution to the JOIN problem for server f in state S

Theorem 5.1

If S is a state consistent with request structure $(\mathcal{R}, \prec_{\mathcal{R}})$, and if f is a server recovering in state S, then state T as constructed above correctly solves the JOIN problem for server f in state S under request structure $(\mathcal{R}, \prec_{\mathcal{R}})$.

Proof: We must show that the five conditions (JC1-JC5) of the JOIN problem are satisfied by state T.

The first condition, JC1 (the consistency of $(\mathcal{L}_{T/f}, \rightarrow_{T/f})$ with $(\mathcal{R}, \prec_{\mathcal{R}})$), follows immediately from the fact that $(\mathcal{L}_{S/f}, \rightarrow_{S/f})$ is consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$ (by premise) and that both log transformations preserve consistency (theorems 4.3 and 4.4).

The second condition, JC2 (the consistency of $(\mathcal{L}_{T/f}, \rightarrow_{T/f})$ with the current states of active objects), follows immediately from the test in step JS1 of the JOIN solution.

Conditions JC3, JC4, and JC5 follow directly from steps JS2, JS3, and JS4 of the JOIN solution, respectively. □

5.2 ACTIVATE Solution

Once a server completes its JOIN phase, it begins recovering its replicas of inactive objects. All recovering servers of an inactive object participate in the object's recovery. The recovering servers start by merging their logs to form the most up-to-date state possible for the object. We let $\mathcal{IS}_{S/A}$ denote this *ideal state* for inactive object A in state S.

$$\mathcal{IS}_{S/A} = \bigcup_{f \in \mathcal{REC}_{S/A}} (\mathcal{L}_{S/f}, \rightarrow_{S/f}) \mid_A$$

The ideal state may be inconsistent with the states of some active objects in the system, however. There may be requests in the ideal state that have dependencies on requests that are not reflected in their objects' active states. These inconsistent requests should be omitted from the new state of object A so that the overall state of the system remains consistent. We let $\mathcal{SAFE}_S(x.A)$ denote the predicate that all of the dependents of request $x.A$, on objects that are active in state S, are present in their objects' active states.

$$\mathcal{SAFE}_S(x.A) \equiv \bigwedge_{\{\ B \in \mathcal{OBJS}\ \mid\ \mathcal{ACT}_{S/B} \neq \emptyset\ \}} \mathcal{DEP}_B(x.A) \subseteq \mathcal{AS}_{S/B}$$

Because we are assuming that explicit dependency information is not available in the system, the exact value of $\mathcal{SAFE}_S(x.A)$ is not available to the recovery mechanism. Instead, we assume that the recovery mechanism has available to it an estimate, $\overline{\mathcal{SAFE}}_S(x.A)$, of the safety predicate. This estimate, like the other estimates, has the property that it is sound.

$$\overline{\mathcal{SAFE}}_S(x.A) \implies \mathcal{SAFE}_S(x.A)$$

The state recovered by the servers of object A will then consist of the requests in the ideal state, $\mathcal{IS}_{S/A}$, that are estimated to be safe. We let $\mathcal{NS}_{S/A}$ denote this state.

$$\mathcal{NS}_{S/A} = \{\ x.A \in \mathcal{IS}_{S/A} \mid \overline{\mathcal{SAFE}}_S(x.A)\ \}$$

Each recovering server installs the *new state* for object A into its log the same way it installed the active states of objects during its JOIN phase. First, the server deletes from its log any request on object A that is not part of the new state. We let $\mathcal{NR}_{S/f}(A)$ denote the set of requests removed from the log of server $f \in \mathcal{REC}_{S/A}$.

$$\mathcal{NR}_{S/f}(A) = (\mathcal{L}_{S/f}, \rightarrow_{S/f}) \mid_A - \mathcal{NS}_{S/A}$$

The server then adds to its log any request in the new state that is not already logged. We let $\mathcal{MS}_{S/f}(A)$ denote the set of requests added to the log of server $f \in \mathcal{REC}_{S/A}$.

$$\mathcal{MS}_{S/f}(A) = \mathcal{NS}_{S/A} - (\mathcal{L}_{S/f}, \rightarrow_{S/f}) \mid_A$$

AS1. $(\mathcal{L}_{T/f}, \rightarrow_{T/f}) = \mathbf{add}_{\mathcal{MS}_{S/f}(A)} (\mathbf{delete}_{\mathcal{NR}_{S/f}(A)} (\mathcal{L}_{S/f}, \rightarrow_{S/f}))$ $\quad \forall f \in \mathcal{REC}_{S/A}$

if $\exists f \in \mathcal{REC}_{S/A}$ *s.t.* $(\mathcal{L}_{T/f}, \rightarrow_{T/f}) |_A \neq \mathcal{NS}_{S/A}$
then abort

if $\exists f \in \mathcal{REC}_{S/A}$ **and** $\exists B \in \mathcal{OBJS}_f \; (f \in \mathcal{ACT}_{S/B})$
s.t. $(\mathcal{L}_{T/f}, \rightarrow_{T/f}) |_B \neq (\mathcal{L}_{S/f}, \rightarrow_{S/f}) |_B$
then abort

if $\exists B \in \mathcal{OBJS} \; (\mathcal{ACT}_{S/B} \neq \emptyset)$ **and** $\exists y.B \in \mathcal{AS}_{S/B}$
s.t. $\overline{\mathcal{DEP}}_A(y.B) \not\subseteq \mathcal{NS}_{S/A}$
then abort

AS2. $(\mathcal{L}_{T/g}, \rightarrow_{T/g}) = (\mathcal{L}_{S/g}, \rightarrow_{S/g}) \qquad \forall g \in \mathcal{SERV} - \mathcal{REC}_{S/A}$

AS3. $\mathcal{ACT}_{T/A} = \mathcal{REC}_{S/A}$
$\mathcal{REC}_{T/A} = \emptyset$

$$\left.\begin{array}{l} \mathcal{ACT}_{T/B} = \mathcal{ACT}_{S/B} \\ \mathcal{REC}_{T/B} = \mathcal{REC}_{S/B} \end{array}\right. \qquad \forall B \in \mathcal{OBJS} - \{A\}$$

AS4. $\mathcal{FAIL}_{T/A} = \mathcal{FAIL}_{S/A} \qquad \forall A \in \mathcal{OBJS}$

Figure 5.2: Solution to the ACTIVATE problem for object A in state S

The complete algorithm for solving the ACTIVATE problem for object A in state S is shown in figure 5.2. Again, T is the state constructed to solve the problem.

Note that the new logs of the recovering servers are tested in step AS1 to make sure that the logged states of active objects are not corrupted. As with the JOIN algorithm, the use of dependency estimates can cause the log transformations to inadvertently add or delete requests from the logged state of an active object. When this occurs, the ACTIVATE algorithm must abort and wait until better dependency estimates can be formed before trying to ACTIVATE object A.

The ACTIVATE algorithm is formally proved correct below:

Theorem 5.2

If S is a state consistent with request structure $(\mathcal{R}, \prec_{\mathcal{R}})$, and if $A \in \mathcal{OBJS}$ is an inactive object in state S, then state T as constructed above correctly solves the ACTIVATE problem for object A in state S under request structure $(\mathcal{R}, \prec_{\mathcal{R}})$.

Proof: We must show that the seven conditions (AC1-AC7) of the ACTIVATE problem are satisfied by state T.

The first condition, AC1 (the consistency of the recovering servers' new logs with $(\mathcal{R}, \prec_{\mathcal{R}})$), follows immediately from the fact that the logs were consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$ in state S (by premise) and that both log transformations preserve consistency (theorems 4.3 and 4.4).

The property that all recovering servers of object A agree on the new state for A (condition AC2) follows directly from the first test in step AS1; if the algorithm does not abort, the logs of all recovering servers of A will reflect $\mathcal{NS}_{S/A}$.

Condition AC3 asserts that the new state for A is consistent with the states of all other active objects in the system. We show that this condition holds in state T in two parts. First, we show that there are no requests in the new state of A that have dependencies on requests that are not part of their objects' active states.

$$\forall x.A \in \mathcal{AS}_{T/A} : \ \forall y.B \in \mathcal{R}\ (y.B \prec_{\mathcal{R}} x.A) : \ \mathcal{ACT}_{T/B} \neq \emptyset \implies y.B \in \mathcal{AS}_{T/B}$$

This portion of the condition follows directly from the definition of safety and the fact that only safe requests are included in the new state of object A. Note that by definition of the ACTIVATE solution, the states of all active objects other than A do not change between states S and T.

The second part of the proof of condition AC3 involves showing that all object A dependents, of requests reflected in the state of another active object, B, are present in the new state of A.

$$\forall y.B \in \mathcal{AS}_{T/B}\ (\mathcal{ACT}_{T/B} \neq \emptyset) : \ \forall x.A \in \mathcal{R}\ (x.A \prec_{\mathcal{R}} y.B) : \ x.A \in \mathcal{AS}_{T/A}$$

This part follows immediately from the third test in step AS1.

Condition AC4 follows immediately from the second test in step AS1 of the algorithm. Conditions AC5, AC6, and AC7 follow immediately from steps AS2, AS3, and AS4 of the algorithm, respectively. □

5.3 Using Explicit Dependency Information

The preceding recovery algorithms assume that explicit dependency information is not available in the system. Both algorithms use estimates of the dependencies between requests to ensure that a recovering server restores consistent states to its object replicas. However, the use of inaccurate estimates sometimes cause the log transformations used by the algorithms to corrupt the logged states of active objects. The algorithms must therefore test for this condition and abort if it occurs.

In this section, we examine how the recovery algorithms are simplified when exact dependency information *is* available in the system. When such information is present, the algorithms can substitute the log transformations based on estimates with those based on exact dependency values. These *precise* transformations have the advantage that they do not corrupt the logged states of active objects. As a result, most of the tests in steps JS1 and AS1 of the recovery algorithms can be omitted.

5.3.1 JOIN Simplification

We begin by showing that the states of active objects logged in step JS1 of the JOIN algorithm are never corrupted when the log transformations based on explicit dependency information are used. We do this in two lemmas. The first lemma shows that the deletion transformation never removes from the log any request in the active state of an object. The second lemma proves that the addition transformation never adds to the log a request on an active object that is not in that object's active state. It follows from these two lemmas that the test in step JS1 of the JOIN solution can be omitted when exact dependency information is available in the system.

Lemma 5.1

When explicit dependent information is available, the deletion transformation in step JS1 of the JOIN recovery algorithm never causes the algorithm to abort.

Proof: We must show that the deletion transformation never removes from a server's log any request that is in the active state of an object. The proof is by contradiction.

Let $f \in \mathcal{SERV}$ be a server recovering in some observably consistent state, S, of the system. Suppose that during the JOIN phase of server f the deletion transformation, $\mathbf{delete}_{\mathcal{NR}_{S/f}}$, removes from the log of server f some request, $x.A$, that is in the active state of object A.

$$x.A \in \mathcal{AS}_{S/A}$$

By definition of $\mathcal{NR}_{S/f}$, we know that $x.A \notin \mathcal{NR}_{S/f}$ because $x.A$ is in the active state of A. Request x.A must therefore have been removed from the log because it depends on some request, $y.B$, in $\mathcal{NR}_{S/f}$.

$$y.B \prec_{\mathcal{R}} x.A$$

However, in order for request $y.B$ to be a member of $\mathcal{NR}_{S/f}$, it must be the case that object B is active in state S and that $y.B$ is not in the active state of B.

$$y.B \notin \mathcal{AS}_{S/B}$$

State S therefore reflects a request, $x.A$, in the active state of an object, A, without reflecting one its dependents, $y.B$, on another active object, B.

$$\begin{array}{ll} \mathcal{ACT}_{S/B} \neq \emptyset & \mathcal{ACT}_{S/A} \neq \emptyset \\ y.B \notin \mathcal{AS}_{S/B} & x.A \in \mathcal{AS}_{S/A} \end{array}$$
$$y.B \prec_{\mathcal{R}} x.A$$

State S is therefore observably inconsistent, a contradiction. The deletion transformation must then have preserved the active states logged for active objects. □

Lemma 5.2

When explicit dependent information is available, the addition transformation in step JS1 of the JOIN recovery algorithm never causes the algorithm to abort.

Proof: We must show that the addition transformation never adds to a server's log any request that is not in the active state of an object. The proof is by contradiction.

Let $f \in \mathcal{SERV}$ be a server recovering in some observably consistent state, S, of the system. Suppose that during the JOIN phase of server f the addition transformation, $\mathbf{add}_{\mathcal{MS}_{S/f}}$, adds to the log of server f some request, $x.A \in \mathcal{R}$, that is not in the active state of object A.

$$x.A \notin \mathcal{AS}_{S/A}$$

By definition of $\mathcal{MS}_{S/f}$, we know that $x.A \notin \mathcal{MS}_{S/f}$ because $x.A$ is not in the active state of A. Request x.A must therefore have been added to the log because it is a dependent of some request, $y.B$, in $\mathcal{MS}_{S/f}$.

$$x.A \prec_{\mathcal{R}} y.B$$

However, in order for request $y.B$ to be a member of $\mathcal{MS}_{S/f}$, it must be the case that object B is active in state S and that $y.B$ is in the active state of B.

$$y.B \in \mathcal{AS}_{S/B}$$

State S therefore reflects a request, $y.B$, in the active state of an object, B, without reflecting one its dependents, $x.A$, on another active object, A.

$$\begin{array}{ll} \mathcal{ACT}_{S/A} \neq \emptyset & \mathcal{ACT}_{S/B} \neq \emptyset \\ x.A \notin \mathcal{AS}_{S/A} & y.B \in \mathcal{AS}_{S/B} \end{array}$$

$$x.A \prec_{\mathcal{R}} y.B$$

State S is therefore observably inconsistent, a contradiction. The addition transformation must then have preserved the active states logged for active objects. □

5.3.2 ACTIVATE Simplification

We now show that the log transformations in step AS1 of the ACTIVATE algorithm do not corrupt the logged states of active objects when exact dependency information is available. Because exact dependency information is available, we assume that the new state, $\mathcal{NS}_{S/A}$, for the object being activated is constructed using the true definition of safety and not an estimate.

Activated Object

We begin by showing that the transformations always correctly install, at the recovering servers, the new state of the object begin activated. This is done in two lemmas analogous to those in the preceding sub-section. It follows from these lemmas that the first test in step AS1 of the ACTIVATE algorithm can be omitted when exact dependency information is available.

Lemma 5.3

When explicit dependency information is available, the deletion transformation in step AS1 of the ACTIVATE recovery algorithm never corrupts the new state logged for the object being activated.

Proof: We must show that the deletion transformation never removes from a recovering server's log any request that is in the new state for the object being activated. The proof is by contradiction.

Let S be an observably consistent state in which some object, $A \in \mathcal{R}$, is being activated. Suppose that during the ACTIVATE phase at some server, f ($f \in \mathcal{REC}_{S/A}$), the deletion transformation $\mathbf{delete}_{\mathcal{NR}_{S/f}(A)}$ removes from the log of server f some request, $x.A$, that is in the new state for object A.

$$x.A \in \mathcal{NS}_{S/A}$$

Because $x.A$ is in $\mathcal{NS}_{S/A}$, it cannot be in $\mathcal{NR}_{S/f}(A)$. Request $x.A$ must therefore have been removed from the log because it depends on some request, $y.A$, in $\mathcal{NR}_{S/f}(A)$.

$$y.A \prec_{\mathcal{R}} x.A$$

Further, because request $y.A$ is in $\mathcal{NR}_{S/f}(A)$, it cannot be in $\mathcal{NS}_{S/A}$.

$$y.A \notin \mathcal{NS}_{S/A}$$

Now, request $y.A$ must be in $\mathcal{IS}_{S/A}$ because it is in $(\mathcal{L}_{S/f}, \rightarrow_{S/f})$ (the log of a recovering server of object A). To see that $y.A$ is in $(\mathcal{L}_{S/f}, \rightarrow_{S/f})$, note that request $x.A$ is in $(\mathcal{L}_{S/f}, \rightarrow_{S/f})$ and so, by definition of consistency, the log must also contain all of the object A dependents of $x.A$, including $y.A$.

Because $y.A$ is in $\mathcal{IS}_{S/A}$ but not in $\mathcal{NS}_{S/A}$, it must be unsafe (by definition of $\mathcal{NS}_{S/A}$). Because $y.A$ is a dependent of $x.A$, request $x.A$ must also be unsafe. However, $x.A$ is included in $\mathcal{NS}_{S/A}$, contradicting the fact that $\mathcal{NS}_{S/A}$ contains only safe requests.

The deletion transformation must therefore have preserved the new logged state for object A. □

Lemma 5.4

When explicit dependency information is available, the addition transformation in step AS1 of the ACTIVATE recovery algorithm never corrupts the new state logged for the object being activated.

Proof: We must show that the addition transformation never adds to a recovering server's log any request, on the object being activated, that is not in that object's new state. The proof is by contradiction.

Let S be an observably consistent state in which some object, $A \in \mathcal{R}$, is being activated. Suppose that during the ACTIVATE phase at some server, f ($f \in \mathcal{REC}_{S/A}$), the addition transformation $\mathbf{add}_{\mathcal{MS}_{S/f}(A)}$ adds to the log of server f some request, $x.A$, that is not in the new state ($\mathcal{NS}_{S/A}$) for object A.

$$x.A \notin \mathcal{NS}_{S/A}$$

Because $x.A$ is not in $\mathcal{NS}_{S/A}$, it cannot be in $\mathcal{MS}_{S/f}(A)$. Request $x.A$ must therefore have been added to the log because it is a dependent of some request, $y.A$, in $\mathcal{MS}_{S/f}(A)$.

$$x.A \prec_{\mathcal{R}} y.A$$

Further, because request $y.A$ is in $\mathcal{MS}_{S/f}(A)$, it must also be in $\mathcal{NS}_{S/A}$.

$$y.A \in \mathcal{NS}_{S/A}$$

We now show that request $x.A$ is unsafe. To see this, first note that request $x.A$ must be in the log of some recovering server of object A. This follows from the fact that $y.A$ is in the log of some recovering server, $g \in \mathcal{REC}_{S/A}$, of object A (because $y.A$ is in $\mathcal{NS}_{S/A}$ and therefore also in $\mathcal{IS}_{S/A}$, which is formed by merging the logs of the recovering object A servers) and from the fact that the log of server g is consistent, and so must contain all of the object A dependents of $y.A$, including $x.A$.

Now, because $x.A$ is in $(\mathcal{L}_{S/g}, \rightarrow_{S/g})$ (the log of a recovering server of A), it must be in $\mathcal{IS}_{S/A}$. However, $x.A$ was omitted from $\mathcal{NS}_{S/A}$. The only reason this could happen is because $x.A$ is unsafe.

Because request $x.A$ is unsafe, and request $y.A$ depends on $x.A$, request $y.A$ must also be unsafe. However, $y.A$ is included in $\mathcal{NS}_{S/A}$, contradicting the fact that $\mathcal{NS}_{S/A}$ only contains safe requests.

The addition transformation must therefore have preserved the new logged state for object A. □

Other Active Objects

We now show that the logged states of other active objects at the recovering servers are not corrupted by the log transformations. Again, we do this in two lemmas. It follows from these lemmas that the second test in step AS1 of the ACTIVATE algorithm is unnecessary when exact dependency information is available.

Lemma 5.5

> *When explicit dependency information is available, the deletion transformation in step AS1 of the ACTIVATE recovery algorithm never corrupts the logged state of any previously active object.*

Proof: Let S be an observably consistent state in which some object, $A \in \mathcal{R}$, is being activated. And, let B denote any other active object in state S. We must show that for any recovering server, f, of object A, if f is an active server of B ($f \in \mathcal{REC}_{S/A} \cap \mathcal{ACT}_{S/B}$) then the deletion transformation does not remove from f's log any request on object B.

The proof is by contradiction. Suppose that the deletion transformation $\mathbf{delete}_{\mathcal{NR}_{S/f}(A)}$ removes from the log of server f some request, $y.B$, on object B. We show that state S would then be observably inconsistent.

Because S is observably consistent, all active servers of B is state S, including f, reflect the active state of B. Because $y.B$ is reflected in the log of f, it follows that $y.B$ is part of the active state of B.

$$y.B \in \mathcal{AS}_{S/B}$$

In order for the deletion transformation to remove request $y.B$ from the log of server f, $y.B$ must be dependent on some object A request, $x.A$, that is removed from the log.

$$\begin{gathered} x.A \in \mathcal{NR}_{S/f}(A) \\ x.A \prec_{\mathcal{R}} y.B \end{gathered} \tag{5.1}$$

Because $x.A$ is in $\mathcal{NR}_{S/f}(A)$, it cannot be part of the new state of object A.

$$x.A \notin \mathcal{NS}_{S/A}$$

Because $x.A$ is in the log of a recovering server of object A, but not included in the new state of that object, request $x.A$ must be unsafe. That is, request $x.A$ is dependent on some other request (for an active object), $z.C$, that is not part of that object's active state.

$$\begin{gathered} z.C \prec_{\mathcal{R}} x.A \\ z.C \notin \mathcal{AS}_{S/C} \end{gathered} \tag{5.2}$$

By transitivity (from 5.1 and 5.2), request $y.B$ is dependent on request $z.C$. The state of object B (an active object) therefore reflects a request, $y.B$, that is dependent on a request, $z.C$, not reflected in the state of object C (another active object).

$$z.C \prec_{\mathcal{R}} y.B$$

$$z.C \notin \mathcal{AS}_{S/C} \qquad y.B \in \mathcal{AS}_{S/B}$$
$$\mathcal{ACT}_{S/C} \neq \emptyset \qquad \mathcal{ACT}_{S/B} \neq \emptyset$$

This contradicts the original assumption that state S is observably consistent. The deletion transformation could not therefore have removed any object B request from the log of server f. □

Lemma 5.6

When explicit dependency information is available, the addition transformation in step AS1 of the ACTIVATE recovery algorithm never corrupts the logged state of any previously active object.

Proof: Let S be an observably consistent state in which some object, $A \in \mathcal{R}$, is being activated. And, let B denote any other active object in state S. We must show that for any recovering server, f, of object A, if f is an active server of B ($f \in \mathcal{REC}_{S/A} \cap \mathcal{ACT}_{S/B}$) then the addition transformation does not add any object B request to the log of server f.

The proof is by contradiction. Suppose that the addition transformation $\mathbf{add}_{\mathcal{MS}_{S/f}(A)}$ adds to the log of server f some request, $y.B$, on object B. We show that the new state for object A contains an unsafe request.

Because S is observably consistent, all active servers of B in state S, including f, reflect the active state of B. Because $y.B$ is added to the log of f (and so was not originally present in the log), it follows that $y.B$ is not part of the active state of B.

$$y.B \notin \mathcal{AS}_{S/B}$$

Request $y.B$ can only have been added to the log by the addition transformation if $y.B$ is a dependent of some object A request, $x.A$, that was also added to the log.

$$x.A \in \mathcal{MS}_{S/f}(A)$$

$$x.A \prec_{\mathcal{R}} y.B$$

Because $x.A$ is in $\mathcal{MS}_{S/f}(A)$, it is part of the new state of object A.

$$x.A \in \mathcal{NS}_{S/A}$$

The new state for object A ($\mathcal{NS}_{S/A}$) therefore reflects a request, $x.A$, that is dependent on an object B request, $y.B$, that is not reflected in the active state of B (an active object).

$$y.B \prec_{\mathcal{R}} x.A$$

$$y.B \notin \mathcal{AS}_{S/B} \qquad x.A \in \mathcal{NS}_{S/A}$$

$$\mathcal{ACT}_{S/B} \neq \emptyset \qquad \mathcal{ACT}_{S/A} = \emptyset$$

That is, the new state for object A reflects an unsafe request, $x.A$, contradicting the fact that $\mathcal{NS}_{S/A}$ only contains safe requests. The addition transformation could not therefore have added any object B request to the log of server f. □

5.4 Summary

Based on the log transformations of chapter 4, we detailed algorithms for solving the JOIN and ACTIVATE recovery problems. We began by describing algorithms for solving the problems when exact dependency information is not available. These algorithms used dependency estimates to derive consistent object and replica states when a server recovered from a failure. It was proved that these algorithms preserve observable consistency in a system.

Because only estimates of the true request dependencies were used, these algorithms could inadvertently corrupt the logged states of objects. The algorithms therefore had to test for corrupted states and abort if such states occurred.

However, it was shown that when exact dependency information is available to the algorithms, no corruption of logged states occurs. Most of the tests in the recovery algorithms could then be omitted when such information is available.

Chapter 6

Estimating Dependencies

When explicit dependency information is not available in a system, the recovery algorithms of chapter 5, as well as the log transformations on which they depend, can use estimates of the dependencies between requests. However, in order to guarantee that consistency is preserved in a system, the algorithms require that the estimates used are always sound. In this chapter we present several dependency estimates having this property.

The estimates are divided into two classes: basic and compound. *Basic estimates* are simple estimates designed to approximate the set of direct dependencies between requests.

Definition 6.1

A dependency between two requests, $x.A \prec_{\mathcal{R}} y.B$, under a request structure $(\mathcal{R}, \prec_{\mathcal{R}})$, is said to be direct if there is no intervening request, $z.C$, through which $x.A$ and $y.B$ are related. Formally,

$$\nexists\, z.C \in \mathcal{R}\ (z.C \neq x.A \bigwedge z.C \neq y.B): \quad x.A \prec_{\mathcal{R}} z.C \ \bigwedge\ z.C \prec_{\mathcal{R}} y.B$$

The basic estimates are formed by examining individual logs for evidence of request orderings. *Compound estimates* are more complicated estimates designed to approximate the set of transitive dependencies between requests.

Definition 6.2

A dependency between two requests, $x.A \prec_{\mathcal{R}} y.B$, under a request structure $(\mathcal{R}, \prec_{\mathcal{R}})$, is said to be transitive if it is not direct.

The compound estimates are formed by combining the results of the basic estimates in order to derive indirect (transitive) dependencies between requests.

6.1 Potential Dependencies

Although we do not assume that the recovery mechanism is given any explicit information about the dependencies between requests, we do assume that it is given some general information about potential dependencies between objects. In particular, we assume that the recovery mechanism has access to a potential dependency relation.

Definition 6.3

A potential dependency relation, $\rightsquigarrow_{\mathcal{R}}$, over request structure $(\mathcal{R}, \prec_{\mathcal{R}})$, is a binary relation on the objects in $\mathcal{OBJS}$ with the property that it relates all pairs of objects between which direct dependencies hold.

$$\forall\, x.A, y.B \in \mathcal{R}: \quad \mathit{direct}\; x.A \prec_{\mathcal{R}} y.B \implies A \rightsquigarrow_{\mathcal{R}} B$$

A potential dependency relation is only an approximation of the direct dependencies that may hold between the states of objects. A potential dependency relation may relate objects between which dependencies do not hold.

$$A \rightsquigarrow_{\mathcal{R}} B \;\;\not\Longleftarrow\;\; \exists\, x.A, y.B \in \mathcal{R}: \; x.A \prec_{\mathcal{R}} y.B$$

The accuracy with which a potential dependency relation reflects the actual dependencies between objects is determined by the application's programmer, who is responsible for providing the recovery mechanism with the potential dependency relation it uses. The programmer should provide the recovery mechanism with the best potential dependency relation that they can construct, based on their knowledge of the application's semantics. In the worst case, the programmer will be unable to determine which objects will be related and so produces a potential dependency relation in which all objects are potentially related. We will use the notation $\rightsquigarrow^*_{\mathcal{R}}$ to refer to the transitive closure of a potential dependency relation $\rightsquigarrow_{\mathcal{R}}$.

In order to help ensure that each direct dependency in an application is represented in the order of requests within some log of the system, the server sets of potentially related objects are restricted so that they overlap.

Overlap Restriction

$$\forall\, A, B \in \mathcal{OBJS}: \;\; A \rightsquigarrow_{\mathcal{R}} B \implies \mathcal{SERV}_A \cap \mathcal{SERV}_B \neq \emptyset$$

There is therefore a tradeoff between the accuracy of a potential dependency relation and the structural restrictions placed on the server sets: any extraneous dependency reflected in the potential dependency relation forces the server sets of the objects involved to unnecessarily overlap. In order to maximize the flexibility of the system structure, it is important that the application's programmer provides the most accurate potential dependency relation possible.

As an example, consider a system containing three objects: A, B, and C. Suppose that an application runs under the following request structure:

$$\text{Request Structure: } (\mathcal{R}, \prec_{\mathcal{R}})$$
$$\mathcal{R} = \{x.A,\; y.B,\; z.C\}$$
$$x.A \prec_{\mathcal{R}} y.B$$

Figure 6.1 depicts three potential dependency relations that are consistent with this request structure. Only potential dependency relation (c) accurately reflects the request structure of the application.

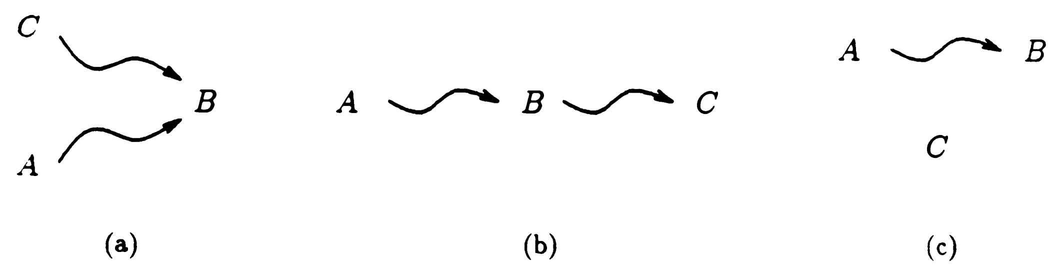


Figure 6.1: Three consistent potential dependency relations

6.2 Basic Estimates

Because the orders of requests in servers' logs are consistent with the request structure of an application, these orders can provide information about the dependencies between requests. The basic dependency estimates are designed to search servers' logs for such information. We begin this section by detailing an estimate for determining when two requests are *not* dependent. This estimate is then used to construct another estimate for determining a request's set of causal dependents.

We assume that when a server fails, all information located at that server becomes inaccessible to the rest of the system. As a result, the recovery mechanism can only use information present in the logs of *functioning servers* (non-failed servers) when constructing dependency estimates.

Definition 6.4

The set of functioning servers of object A in state S are:

$$\mathcal{FUNC}_{S/A} = \mathcal{ACT}_{S/A} \bigcup \mathcal{REC}_{S/A}$$

6.2.1 Request Ordering

The causal consistency condition on logs guarantees that when a server logs some request, $y.B$, it has previously logged all requests (on objects with replicas managed by the server) on which $y.B$ depends. It follows then that if a server logs request $x.A$ *after* request $y.B$, then request $y.B$ cannot be dependent on request $x.A$. Further, if a server of objects A and B logs $y.B$ without logging $x.A$, then request $y.B$ cannot be dependent on $x.A$.

In addition, the observable consistency condition on states guarantees that if a request, $y.B$, is reflected in the active state of an object, B, then any request on which it is dependent, $x.A$, is reflected in the active state of its object, A (provided object A is active). It follows that if both objects A and B are active, and $y.B$ is reflected in the active state of B but $x.A$ is not reflected in the active state of A, then request $y.B$ is not dependent on request $x.A$.

Combining this intuition along with the dependency information provided by the potential dependency relation, we can estimate when two requests ($x.A$ and $y.B$) are *not* related. We let $\mathbf{con}^0_S(x.A \prec y.B)$ denote this basic estimate.

Definition 6.5

Let $(\mathcal{R}, \prec_{\mathcal{R}})$ be a request structure, let $\rightsquigarrow_{\mathcal{R}}$ be a potential dependency relation consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$, and let S be a system state consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. The request ordering, $x.A \prec y.B$, is directly contradicted in state S, denoted $\mathbf{con}^0_S(x.A \prec y.B)$, if any of the following four conditions holds:

1. $A \not\rightsquigarrow^*_{\mathcal{R}} B$
2. $\exists\, f \in \mathcal{FUNC}_{S/A} \cap \mathcal{FUNC}_{S/B} :\ x.A, y.B \in \mathcal{L}_{S/f} \wedge y.B \rightarrow_{S/f} x.A$
3. $\exists\, f \in \mathcal{FUNC}_{S/A} \cap \mathcal{FUNC}_{S/B} :\ y.B \in \mathcal{L}_{S/f} \wedge x.A \notin \mathcal{L}_{S/f}$
4. $\mathcal{ACT}_{S/A} \neq \emptyset \wedge \mathcal{ACT}_{S/B} \neq \emptyset \wedge x.A \notin \mathcal{AS}_{S/A} \wedge y.B \in \mathcal{AS}_{S/B}$

This estimate has the property that it is sound. When an ordering, $x.A \prec y.B$, is found to be directly contradicted, it is guaranteed that $y.B$ is not dependent on $x.A$. However, if the ordering is not found to be contradicted, the requests may or may not be ordered.

Theorem 6.1

For any request structure $(\mathcal{R}, \prec_{\mathcal{R}})$, potential dependency relation $\leadsto_{\mathcal{R}}$ consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$, system state S consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$, and pair of requests $x.A, y.B \in \mathcal{R}$:

$$\mathbf{con}^0_S(x.A \prec y.B) \implies x.A \not\prec_{\mathcal{R}} y.B$$

Proof: The proof is by contradiction. Suppose that requests $x.A$ and $y.B$ are related ($x.A \prec_{\mathcal{R}} y.B$), but that the order is found to be directly contradicted ($\mathbf{con}^0_S(x.A \prec y.B)$).

Because the order is directly contradicted, at least one of the four conditions in the estimate definition must hold. If the first condition holds ($A \not\leadsto^*_{\mathcal{R}} B$), then the potential dependency relation is inconsistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. If either the second or third condition holds, then the log of server f is inconsistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. Finally, if the fourth condition holds, then the system state is observably inconsistent with $(\mathcal{R}, \prec_{\mathcal{R}})$.

In either case, an inconsistency would exist in the system (contradicting the assumption that the system is consistent) and so the theorem assertion must hold. □

As an example, consider the system shown in figure 6.2. Depicted are the logs of two servers, f and g, along with a potential dependency relation. Server f manages replicas of objects A and B, while server g manages replicas of objects

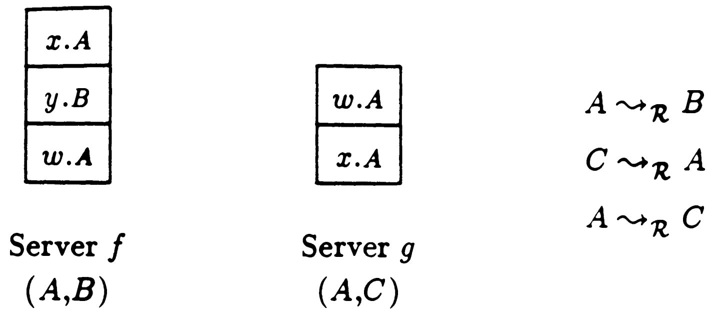


Figure 6.2: An example of direct contradiction

Table 6.1: Directly Contradicted Request Orderings

Condition 1	Condition 2	Condition 3
$y.B \prec w.A$	$w.A \prec x.A$	$z.C \prec w.A$
$y.B \prec x.A$	$y.B \prec x.A$	$z.C \prec x.A$
$y.B \prec z.C$	$w.A \prec y.B$	
$x.A \prec w.A$		

A and C. Suppose that in addition to those requests present in the logs, the system also contains a fourth request, $z.C$, on object C. Table 6.1 summarizes the request orderings that are directly contradicted by this system, if all objects are inactive. The orderings are broken down according to the conditions of the estimate definition that caused them to be contradicted. Note that the following orderings are not directly contradicted anywhere in the system:

$$w.A \prec z.C \qquad x.A \prec y.B \qquad x.A \prec z.C \qquad z.C \prec y.B$$

6.2.2 Dependency Set

Using the preceding estimate, we can now construct an estimate of $\mathcal{DEP}_B(x.A)$, the object B dependents of request $x.A$. Again, this estimate is based on the consistency restrictions placed on logs and system states.

From the causal consistency condition on logs, we know that if a server of objects A and B logs request $x.A$, then it previously has logged all of the object B dependents of $x.A$. The set of object B requests preceding $x.A$ in a log can therefore be used as an estimate of the true set of dependents. From the observable consistency condition on system states, we know that if both objects A and B are active, and the active state of A reflects request $x.A$, then the active state of B must reflect all of the object B dependents of $x.A$. In this case, the set of requests in the active state of B can also be used as an estimate of the dependency set.

Of course, not all of the object B requests in these estimates may be dependents of $x.A$. There may be information in the system that contradicts the ordering between $x.A$ and some of the object B requests. This information can be used to further refine the estimates.

Definition 6.6

Let $(\mathcal{R}, \prec_{\mathcal{R}})$ be a request structure, let $\leadsto_{\mathcal{R}}$ be a potential dependency relation consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$, and let S be a system state observably consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. For any object $B \in \mathcal{OBJS}$ and request $x.A \in \mathcal{R}$, the basic estimated dependents of $x.A$ are:

$$\mathbf{dep}^0_{S/B}(x.A) = \begin{cases} \bot & \textit{if } \neg\exists f \in \mathcal{FUNC}_{S/A} \cap \mathcal{FUNC}_{S/B} : \ x.A \in \mathcal{L}_{S/f} \\ & \qquad\qquad \textit{and} \\ & \quad \mathcal{ACT}_{S/A} = \emptyset \ \vee \ \mathcal{ACT}_{S/B} = \emptyset \ \vee \ x.A \notin \mathcal{AS}_{S/A} \\ \emptyset & \textit{if } B \not\leadsto^*_{\mathcal{R}} A \\ \{y.B \mid \neg\mathbf{con}^0_S(y.B \prec x.A) \ \wedge & \qquad\qquad o.w. \\ \quad [\ \exists f \in \mathcal{FUNC}_{S/A} \cap \mathcal{FUNC}_{S/B} : x.A, y.B \in \mathcal{L}_{S/f} & \\ \quad \vee \ y.B \in \mathcal{AS}_{S/B} \ ]\ \} & \end{cases}$$

Like the first basic estimate, the dependency set estimate has the property that it is sound.

Theorem 6.2

Let $(\mathcal{R}, \prec_{\mathcal{R}})$ be any request structure, $\rightsquigarrow_{\mathcal{R}}$ be any potential dependency relation consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$, and S be any system state observably consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. For any request $x.A \in \mathcal{R}$ and object $B \in \mathcal{OBJS}$, if $\mathbf{dep}^0_{S/B}(x.A)$ is defined then:

$$\mathcal{DEP}_B(x.A) \subseteq \mathbf{dep}^0_{S/B}(x.A)$$

Proof: The proof is by contradiction. Suppose that $\mathbf{dep}^0_{S/B}(x.A)$ is defined, but that there exists some dependent, $y.B$, of request $x.A$ that is not included in $\mathbf{dep}^0_{S/B}(x.A)$.

$$y.B \in \mathcal{DEP}_B(x.A) \qquad y.B \notin \mathbf{dep}^0_{S/B}(x.A)$$

There are three conditions under which $\mathbf{dep}^0_{S/B}(x.A)$ is defined:

Case 1: $B \not\rightsquigarrow^*_{\mathcal{R}} A$

In this case, the potential dependency relation does not reflect the real dependency between $x.A$ and $y.B$, and so is inconsistent with the request structure of the application. This contradicts the assumption that the potential dependency relation is consistent.

Case 2: $B \rightsquigarrow^*_{\mathcal{R}} A \;\wedge\; \exists\, f \in \mathcal{FUNC}_{S/A} \cap \mathcal{FUNC}_{S/B} : x.A \in \mathcal{L}_{S/f}$

Because the log of server f contains request $x.A$, and because the state of the system is causally consistent, the log of server f must also contain request $y.B$.

$$y.B \in \mathcal{L}_{S/f}$$

From the definition of the dependency set estimate, the only reason $y.B$ could then be omitted from the estimate is because the ordering between it and request $x.A$ is directly contradicted somewhere in the system.

$$\mathbf{con}_S^0(y.B \prec x.A) \;\equiv\; \mathbf{true}$$

However, from theorem 6.1, this implies that the two requests are unrelated.

$$y.B \not\prec_{\mathcal{R}} x.A$$

This contradicts the assumption that $y.B$ is a real dependent of $x.A$.

Case 3: $B \leadsto_{\mathcal{R}}^{*} A \;\wedge\; \mathcal{ACT}_{S/A} \neq \emptyset \;\wedge\; \mathcal{ACT}_{S/B} \neq \emptyset \;\wedge\; x.A \in \mathcal{AS}_{S/A}$

Because both objects A and B are active, and the active state of A reflects $x.A$, and because the system state is observably consistent, the active state of B must reflect all of the object B dependents of request $x.A$, including $y.B$.

$$y.B \;\in\; \mathcal{AS}_{S/B}$$

From the definition of the dependency set estimate, the only reason $y.B$ could then be omitted from the estimate is because the ordering between it and request $x.A$ is directly contradicted somewhere in the system.

$$\mathbf{con}_S^0(y.B \prec x.A) \;\equiv\; \mathbf{true}$$

However, from theorem 6.1, this implies that the two requests are unrelated.

$$y.B \not\prec_{\mathcal{R}} x.A$$

This contradicts the assumption that $y.B$ is a real dependent of $x.A$.

In either case, a contradiction occurs and so the original assumption must be incorrect. The estimate must therefore always include all true dependents when

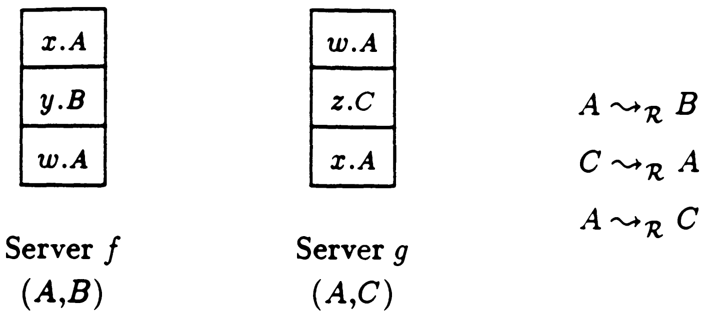


Figure 6.3: A example of basic dependency set estimation

Table 6.2: Basic Estimated Dependents

	$w.A$	$x.A$	$y.B$	$z.C$
A	$\emptyset$	$\emptyset$	$x.A$	$w.A$
B	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
C	$\emptyset$	$z.C$	$\perp$	$\emptyset$

defined. □

As an example, consider the system shown in figure 6.3. This system is identical to the system shown in figure 6.2, except that server g has logged request $z.C$ between requests $w.A$ and $x.A$. For each request in the system, table 6.2 shows the basic estimated dependents on objects A, B, and C.

6.3 Compound Estimates

Requests are not always directly related. Two requests, $x_1.A_1$ and $x_n.A_n$, can be related through a sequence of dependencies on other requests in the system.

$$x_1.A_1 \prec_{\mathcal{R}} x_2.A_2 \prec_{\mathcal{R}} \cdots \prec_{\mathcal{R}} x_n.A_n$$

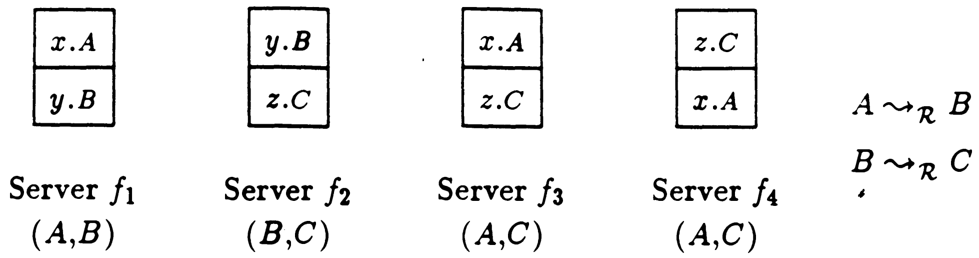
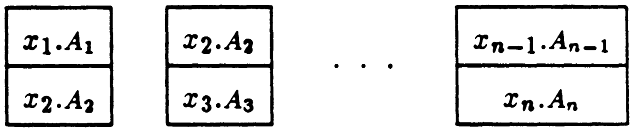


Figure 6.4: Non-optimal transitive closure

The information necessary to detect these transitive dependencies may be embedded across multiple logs in the system. For example, the above transitive dependency might embed itself across $n-1$ logs.



The compound estimates combine the results of the basic estimates in order to detect such transitive dependencies. By combining the results of the basic estimates, the compound estimates are able to approximate the sequences out of which the transitive dependencies are built.

An obvious method for estimating transitive dependencies is to simply take the transitive closure of the basic estimates. This method is not entirely accurate, however. For example, consider the system shown in figure 6.4. This figure depicts a system with four servers (f_1, f_2, f_3, and f_4), three objects (A, B, and C), and three requests ($x.A$, $y.B$, and $z.C$). Applying the basic estimates, we determine that two orderings are possible:

$$x.A \prec_{\mathcal{R}} y.B \qquad y.B \prec_{\mathcal{R}} z.C$$

By taking the transitive closure, we would also estimate that request $z.C$ is dependent on request $x.A$, even though the logs of servers f_3 and f_4 directly contradict any ordering between the two requests. The compound estimates presented in this section detect contradictions, such as the one between requests $z.A$ and $x.A$, and use them to form more accurate approximations when combining the basic estimates.

We refer to the sequence of objects over which a transitive dependency may be embedded as a chain.

Definition 6.7

A chain, H, is a sequence of potentially dependent objects.

$$H = A_1 \rightsquigarrow_{\mathcal{R}} A_2 \rightsquigarrow_{\mathcal{R}} \cdots \rightsquigarrow_{\mathcal{R}} A_n$$

Definition 6.8

A sub-chain of a chain, H,

$$H = A_1 \rightsquigarrow_{\mathcal{R}} A_2 \rightsquigarrow_{\mathcal{R}} \cdots \rightsquigarrow_{\mathcal{R}} A_n$$

is any subsequence of its objects

$$H' = A_{m_1} \rightsquigarrow^*_{\mathcal{R}} A_{m_2} \rightsquigarrow^*_{\mathcal{R}} \cdots \rightsquigarrow^*_{\mathcal{R}} A_{m_p}$$

where $1 \leq m_1 < m_2 < \ldots < m_p \leq n$.

Definition 6.9

The A_iA_j sub-chain of a chain, H, is the sub-chain of objects from A_i to A_j:

$$H_{i..j} = A_i \rightsquigarrow_{\mathcal{R}} A_{i+1} \rightsquigarrow_{\mathcal{R}} \cdots \rightsquigarrow_{\mathcal{R}} A_j$$

Definition 6.10

The length of a chain or sub-chain, H, denoted $\|H\|$, is the number of objects in the sequence.

6.3.1 Dependency Set

In this subsection we present our compound estimate of $\mathcal{DEP}_B(x.A)$, the object B dependents of request $x.A$, which we denote as $\mathbf{dep}^{\omega}_{S/B}(x.A)$. This estimate is constructed by estimating the object B dependents of $x.A$ that occur along each chain from object B to object A, and then combining the results from the different chains.

We begin by describing our estimate of the dependents that occur along a particular chain, H

$$H = A_1 \leadsto_{\mathcal{R}} A_2 \leadsto_{\mathcal{R}} \cdots \leadsto_{\mathcal{R}} A_n$$

For any request, $x_n.A_n$, we let $\mathbf{dep}^{\omega}_{S/H}(x_n.A_n)$ denote our estimate in state S of the object A_1 dependents of $x_n.A_n$ that occur along chain H. This estimate can be formed in many ways, depending up which servers are functioning in state S. First, if there is a functioning server of objects A_1 and A_n that has logged request $x_n.A_n$, the basic estimate can be applied to determine the dependency set. In general, however, the server sets of objects A_1 and A_n will not overlap, unless the objects are directly related.

Alternately, an estimate can be formed by sub-dividing the problem as shown in figure 6.5. First, an object in the chain, A_i $(1 < i < n)$, is selected. Next, the object A_i dependents of $x_n.A_n$ are estimated. Finally, the object A_1 dependents of the object A_i dependents are estimated to produce the desired dependency set. Again, if the server sets of objects A_1 and A_i overlap, and if the server sets of objects A_i and A_n overlap, the basic estimates can be applied to solve each

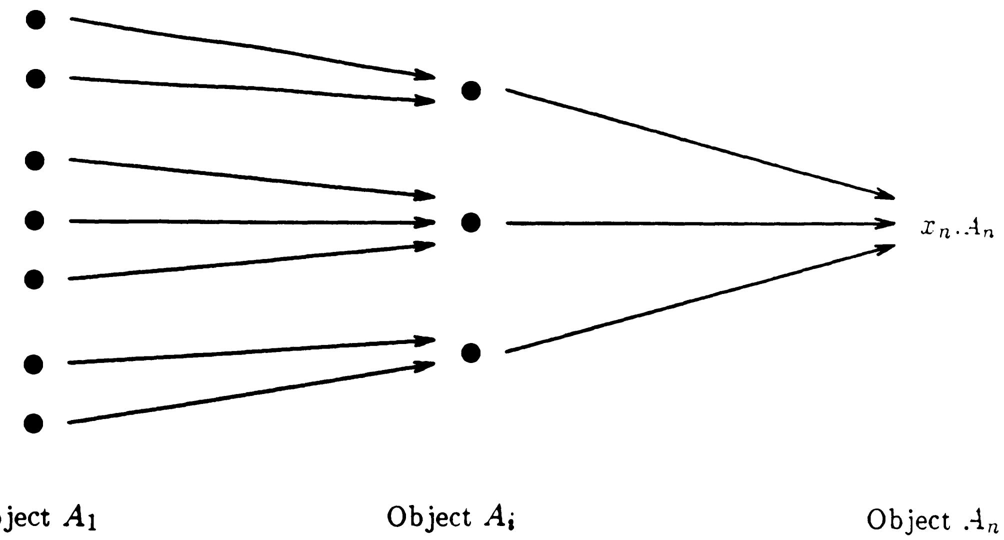


Figure 6.5: Sub-dividing an estimate along a chain

of the sub-problems. The result is a dependency set estimate obtained along the sub-chain:

$$A_1 \leadsto^*_{\mathcal{R}} A_i \leadsto^*_{\mathcal{R}} A_n$$

If the server sets do not overlap, each of the sub-problems must be further sub-divided until the basic estimates can be applied. In general, the problem is sub-divided until a sub-chain of H is found

$$A_1 \leadsto^*_{\mathcal{R}} A_{m_1} \leadsto^*_{\mathcal{R}} A_{m_2} \leadsto^*_{\mathcal{R}} \cdots \leadsto^*_{\mathcal{R}} A_{m_p} \leadsto^*_{\mathcal{R}} A_n$$
$$1 < m_1 < m_2 < \ldots < m_p < n$$

in which each pair of adjacent objects have overlapping server sets.

This procedure is summarized in the following recursive estimate definition. Note that the estimate has been extended to operate on sets of requests. In particular, if Q is a set of object A_n requests, then $\mathbf{dep}^{\omega}_{S/H}(Q)$ denotes the estimated set of object A_1 dependents of the requests in Q.

$$\mathbf{dep}^{\omega}_{S/H}(Q) \;=\; \begin{cases} \bigcup\limits_{x_2.A_2 \in Q} \mathbf{dep}^{0}_{S/A_1}(x_2.A_2) & \text{if defined} \\[2ex] \bigcup\limits_{x_n.A_n \in Q} \mathbf{dep}^{\omega}_{S/H_{1..i}}(\mathbf{dep}^{\omega}_{S/H_{i..n}}(x_n.A_n)) & o.w. \\[2ex] \quad \text{where } 1 < i < n \text{ is chosen so that the estimates} & \\ \quad \text{are defined.} & \end{cases}$$

Note also that the definitions of union and intersection (intersection is used later in this section) must be altered to take into account the possibility of undefined sets.

$$\bigcup_i S_i \;=\; \begin{cases} \perp & \text{if } \exists i: \; S_i = \perp \\ \bigcup\limits_i S_i & \text{o.w.} \end{cases}$$

$$\bigcap_i S_i \;=\; \begin{cases} \perp & \text{if } \forall i: \; S_i = \perp \\ \bigcap\limits_{\{i \mid S_i \neq \perp\}} S_i & \text{o.w.} \end{cases}$$

The choice of object, A_i, at which to sub-divide a problem can affect the final estimate. Different object choices can yield slightly different approximations. When an estimate is defined, though, it is guaranteed to be sound. It follows that an accurate approximation of the dependency set (one with few extraneous requests) can be formed by intersecting the estimates from each of the different sub-division choices. The complete dependency set estimate along chain H is given below.

Definition 6.11

Let $(\mathcal{R}, \prec_{\mathcal{R}})$ be a request structure, let $\leadsto_{\mathcal{R}}$ be a potential dependency relation consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$, and let S be a system state consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. For any chain,

$$H \;=\; A_1 \leadsto_{\mathcal{R}} A_2 \leadsto_{\mathcal{R}} \cdots \leadsto_{\mathcal{R}} A_n$$

and set of object A_n requests, Q, the estimated dependents of Q along chain H are:

$$\mathbf{dep}^{\omega}_{S/H}(Q) \;=\; \begin{cases} \bigcup_{x_2.A_2 \in Q} \mathbf{dep}^{0}_{S/A_1}(x_2.A_2) & \|H\| = 2 \\ \bigcup_{x_n.A_n \in Q} [\mathbf{dep}^{0}_{S/A_1}(x_n.A_n) \cap & \|H\| > 2 \\ \quad [\bigcap_{1<i<n} \mathbf{dep}^{\omega}_{S/H_{1..i}}(\mathbf{dep}^{\omega}_{S/H_{i..n}}(x_n.A_n))\;]\;] & \end{cases}$$

Theorem 6.3

When it is defined, $\mathbf{dep}^{\omega}_{S/H}(Q)$ does not under-estimate the true set of dependencies along chain H.

Proof: Let $x_n.A_n$ denote any request in Q. Suppose that $\mathbf{dep}^{\omega}_{S/H}(Q)$ is defined and that the system contains a transitive dependency along chain H.

$$x_1.A_1 \prec_{\mathcal{R}} x_2.A_2 \prec_{\mathcal{R}} \cdots \prec_{\mathcal{R}} x_n.A_n$$

We show by induction on the length of the chain that $\mathbf{dep}^{\omega}_{S/H}(Q)$ contains $x_1.A_1$.

Base Case: $\|H\| \;=\; 2$

The dependency set estimate is the union of basic estimates.

$$\bigcup_{x_2.A_2 \in Q} \mathbf{dep}^{0}_{S/A_1}(x_2.A_2)$$

By assumption this union is defined, and so each of the component basic estimates must also be defined, including $\mathbf{dep}^{0}_{S/A_1}(x_2.A_2)$. From theorem 6.2,

$\mathbf{dep}^0_{S/A_1}(x_2.A_2)$ contains all object A_1 dependents of request $x_2.A_2$, including $x_1.A_1$. It follows then that request $x_1.A_1$ is included in the union.

Induction Step: $\|H\| = n > 2$

Suppose that the theorem holds for all chains with length less than n.

For a chain of length n, the dependency set estimate is the union of components, each of which in turn is an intersection of estimates. We show that one of these components, specifically the one shown below, contains request $x_1.A_1$. It then follows that the overall union contains $x_1.A_1$.

$$\mathbf{dep}^0_{S/A_1}(x_n.A_n) \bigcap \left[\bigcap_{1<i<n} \mathbf{dep}^\omega_{S/H_{1..i}}(\mathbf{dep}^\omega_{S/H_{i..n}}(x_n.A_n)) \right]$$

In order to show that this component contains the desired request, we show that each element in the intersection (when defined) contains the request. First, consider the estimate $\mathbf{dep}^0_{S/A_1}(x_n.A_n)$. From theorem 6.2, this estimate (when defined) contains all of the object A_1 dependents of request $x_n.A_n$, including $x_1.A_1$.

Now, consider any of the remaining elements, $\mathbf{dep}^\omega_{S/H_{1..i}}(\mathbf{dep}^\omega_{S/H_{i..n}}(x_n.A_n))$, that is defined. By the induction hypothesis, $\mathbf{dep}^\omega_{S/H_{i..n}}(x_n.A_n)$ contains all of the object A_i dependents of request $x_n.A_n$ that occur along chain $H_{i..n}$, including request $x_i.A_i$. Applying the induction hypothesis again, we see that $\mathbf{dep}^\omega_{S/H_{1..i}}(\mathbf{dep}^\omega_{S/H_{i..n}}(x_n.A_n))$ contains all of the object A_1 dependents of $x_i.A_i$ that occur along chain $H_{1..i}$, including $x_1.A_1$.

□

The general estimate of the object B dependents of a request, $x.A$, is formed by unioning the estimated dependents along all chains from B to A. We denote the set of all chains from object B to object A as $BA\text{-}\mathcal{CHAINS}$.

Definition 6.12

Let $(\mathcal{R}, \prec_{\mathcal{R}})$ be a request structure, let $\rightsquigarrow_{\mathcal{R}}$ be a potential dependency relation consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$, and let S be a system state consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. For any object, B, and request, $x.A$, the estimated object B dependents of request $x.A$ are:

$$\mathbf{dep}^{\omega}_{S/B}(x.A) \;=\; \bigcup_{H \in BA\text{-}\mathcal{CHAINS}} \mathbf{dep}^{\omega}_{S/H}(x.A)$$

Theorem 6.4

When it is defined, $\mathbf{dep}^{\omega}_{S/B}(x.A)$ does not under-estimate the true set of dependents.

$$\mathcal{DEP}_B(x.A) \;\subseteq\; \mathbf{dep}^{\omega}_{S/B}(x.A)$$

Proof: By definition, any object B dependent, $y.B$, of request $x.A$ is dependent along some chain, H, from B to A. From theorem 6.3, the estimated dependents along chain H include $y.B$. It follows that any object B dependent of $x.A$ is included in the union. □

6.3.2 Request Ordering

Now consider the problem of estimating when two requests, $x.A$ and $y.B$, are unrelated. We let $\mathbf{con}^{\omega}_{S}(x.A \prec y.B)$ denote our compound estimate of the predicate that request $y.B$ is *not* causally dependent on request $x.A$. This estimate is constructed in a manner similar to the preceding compound estimate. First, the relationship of the two requests is estimated along each chain from object A to object B. The results of the estimates are then combined to form an overall estimate of whether the two requests are related.

We let $\mathbf{con}^{\omega}_{S/H}(x_1.A_1 \prec x_n.A_n)$ denote our estimate of the predicate that request $x_n.A_n$ is not causally dependent on request $x_1.A_1$ along chain H.

$$H = A_1 \rightsquigarrow_{\mathcal{R}} A_2 \rightsquigarrow_{\mathcal{R}} \cdots \rightsquigarrow_{\mathcal{R}} A_n$$

The idea behind the construction of this estimate is to search the chain for an object, A_i, such that none of the object A_i dependents of $x_n.A_n$ are dependent on $x_1.A_1$. The existence of such an object implies that request $x_n.A_n$ is not transitively dependent on request $x_1.A_1$ through a sequence of requests on objects that include A_i. Because H contains A_i, this in turn implies that the requests are not related along chain H.

The estimate is formed by examining each object, A_i, in the chain. For each such object, the dependents of request $x_n.A_n$ are estimated. Each of these dependents is then recursively tested to determine if they are dependent on request $x_1.A_1$. The complete estimate definition is given below. Note that the definition is extended to operate on sets of requests. In particular, if Q is a set of object A_n requests, then $\mathbf{con}^{\omega}_{S/H}(x_1.A_1 \prec Q)$ denotes our estimate of the predicate that none of the requests in Q are dependent on $x_1.A_1$ along chain H.

Definition 6.13

Let $(\mathcal{R}, \prec_{\mathcal{R}})$ be a request structure, let $\rightsquigarrow_{\mathcal{R}}$ be a potential dependency relation consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$, and let S be a system state consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. For any chain,

$$H \;=\; A_1 \rightsquigarrow_{\mathcal{R}} A_2 \rightsquigarrow_{\mathcal{R}} \cdots \rightsquigarrow_{\mathcal{R}} A_n$$

request $x_1.A_1$, and set of object A_n requests Q, the dependency of Q on request $x_1.A_1$ along chain H is <u>contradicted</u> in state S, denoted $\mathbf{con}^{\omega}_{S/H}(x_1.A_1 \prec Q)$, if the following condition holds.

$$\begin{array}{ll} \bigwedge_{x_2.A_2 \in Q} \mathbf{con}^{0}_{S}(x_1.A_1 \prec x_2.A_2) & \textit{if } \; \|H\| = 2 \\ \\ \bigwedge_{x_n.A_n \in Q} [\ \mathbf{con}^{0}_{S}(x_1.A_1 \prec x_n.A_n) \quad \vee & o.w. \\ \quad [\bigvee_{1<i<n} \mathbf{con}^{\omega}_{S/H_{1..i}}(x_1.A_1 \prec \mathbf{dep}^{\omega}_{S/H_{i..n}}(x_n.A_n))\ ]\ ] & \end{array}$$

Theorem 6.5

If $\mathbf{con}^{\omega}_{S/H}(x_1.A_1 \prec Q)$ holds, then there does not exist any request in Q that is dependent on $x_1.A_1$ along chain H.

Proof: The proof is by contradiction. Suppose that $\mathbf{con}^{\omega}_{S/H}(x_1.A_1 \prec Q)$ holds, but that there exists a request, $x_n.A_n$, in Q that is dependent on $x_1.A_1$ through a sequence of dependencies along chain H.

$$x_1.A_1 \prec_{\mathcal{R}} x_2.A_2 \prec_{\mathcal{R}} \cdots \prec_{\mathcal{R}} x_n.A_n$$

We show by induction on the length of chain H that an inconsistency exists.

<u>Base Case:</u> $\|H\| \;=\; 2$

Because request $x_n.A_n$ is dependent on request $x_1.A_1$ ($x_1.A_1 \prec_{\mathcal{R}} x_n.A_n$), we know from theorem 6.1 that $\mathbf{con}^{0}_{S}(x_1.A_1 \prec x_n.A_n)$ is false. Because this is one of the conjuncts in the definition of $\mathbf{con}^{\omega}_{S/H}(x_1.A_1 \prec Q)$, it follows that the compound estimate is false, contradicting the assumption that it's true.

Induction Step: $\|H\| \;=\; n \;>\; 2$

Suppose that the theorem holds for all chains with length less than n. We show that the conjunct, corresponding to request $x_n.A_n$, in the definition of $\mathbf{con}^{\omega}_{S/H}(x_1.A_1 \prec Q)$ is false. It then follows that the overall compound estimate is false, contradicting the the assumption that the estimate is true.

We show that the conjunct is false by showing that each of its disjuncts is false. First, from theorem 6.1 we know that

$$\mathbf{con}^{0}_{S}(x_1.A_1 \prec x_n.A_n) \;\equiv\; \mathbf{false}$$

Now, consider any of the disjuncts $\mathbf{con}^{\omega}_{S/H_{1..i}}(x_1.A_1 \prec \mathbf{dep}^{\omega}_{S/H_{i..n}}(x_n.A_n))$ From theorem 6.3, we know that when it is defined $\mathbf{dep}^{\omega}_{S/H_{i..n}}(x_n.A_n)$ contains all of the object A_i dependents of $x_n.A_n$, including $x_i.A_i$. Because $x_i.A_i$ is dependent on $x_1.A_1$ ($x_1.A_1 \prec_{\mathcal{R}} x_i.A_i$), we now by the induction hypothesis that

$$\mathbf{con}^{\omega}_{S/H_{1..i}}(x_1.A_1 \prec x_i.A_i) \;\equiv\; \mathbf{false}$$

It therefore follows that

$$\mathbf{con}^{\omega}_{S/H_{1..i}}(x_1.A_1 \prec \mathbf{dep}^{\omega}_{S/H_{i..n}}(x_n.A_n)) \;\equiv\; \mathbf{false}$$

□

The general compound estimate of the relationship between two requests, $x.A$ and $y.B$, is formed by combining the estimates of the requests' relationship along individual chains.

Definition 6.14

Let $(\mathcal{R}, \prec_{\mathcal{R}})$ be any request structure, let $\leadsto_{\mathcal{R}}$ be a potential dependency relation consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$, and let S be a system state consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. For any pair of requests, $x.A$ and $y.B$, the dependency of $y.B$ on $x.A$ is <u>*contradicted*</u> *in state S if $\mathbf{con}_S^{\omega}(x.A \prec y.B)$ holds.*

$$\mathbf{con}_S^{\omega}(x.A \prec y.B) \;=\; \bigwedge_{H \in AB\text{-}\mathcal{CHAINS}} \mathbf{con}_{S/H}^{\omega}(x.A \prec y.B)$$

Theorem 6.6

$\mathbf{con}_S^{\omega}(x.A \prec y.B)$ *does not under-estimate the true set of related requests.*

$$\mathbf{con}_S^{\omega}(x.A \prec y.B) \implies x.A \not\prec_{\mathcal{R}} y.B$$

Proof: We show the contrapositive. Suppose that request $y.B$ is causally dependent on request $x.A$ ($x.A \prec_{\mathcal{R}} y.B$). By definition, the two requests are related along some chain, H, from object A to object B. From theorem 6.5, we know that

$$\mathbf{con}_{S/H}^{\omega}(x.A \prec y.B) \equiv \mathbf{false}$$

Because this is one of the conjuncts in the definition of $\mathbf{con}_S^{\omega}(x.A \prec y.B)$, it follows that

$$\mathbf{con}_S^{\omega}(x.A \prec y.B) \equiv \mathbf{false}$$

□

6.3.3 Safety

Our last compound estimate approximates the safety predicate $\mathcal{SAFE}_S(x.A)$. Recall that, when true, the safety predicate indicates that the dependents (on active objects) of request $x.A$ are reflected in their objects' current states. Like the other compound estimates, the safety estimate is formed by combining estimates of safety along individual chains that lead to object A.

For any request $x_n.A_n$, active object A_1 ($\mathcal{ACT}_{S/A_1} \neq \emptyset$), and chain H from object A_1 to object A_n,

$$H = A_1 \leadsto_{\mathcal{R}} A_2 \leadsto_{\mathcal{R}} \cdots \leadsto_{\mathcal{R}} A_n$$

we let $\text{safe}^{\omega}_{S/H}(x_n.A_n)$ denote our estimate of the predicate that all object A_1 dependents of request $x_n.A_n$ (along chain H) are reflected in the active state of A_1. One method for constructing this estimate is to approximate the object A_1 dependents of request $x_n.A_n$ (using one of the preceding estimates) and then check to see if all of those estimated dependents are reflected in the state of A_1. However, this method will only work when the dependency set estimate is defined.

Another method for constructing the estimate is to examine each active object A_i ($\mathcal{ACT}_{S/A_i} \neq \emptyset$) in the chain, estimate the object A_i dependents of request $x_n.A_n$, and then check to see if all of these dependents are reflected in the active state of object A_i. The intuition behind this method is that if $x_n.A_n$ is safe along chain H then all of its object A_i dependents are also safe along chain $H_{1..i}$. If one of these object A_i dependents were unsafe, then it would not be reflected in the active state of A_i, because the state of A_i would be inconsistent with the state of A_1.

Definition 6.15

Let $(\mathcal{R}, \prec_{\mathcal{R}})$ be a request structure, let $\leadsto_{\mathcal{R}}$ be a potential dependency relation consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$, and let S be a system state consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. For any request $x_n.A_n$, active object A_1 ($\mathcal{ACT}_{S/A_1} \neq \emptyset$), and chain H from A_1 to A_n,

$$H = A_1 \leadsto_{\mathcal{R}} A_2 \leadsto_{\mathcal{R}} \cdots \leadsto_{\mathcal{R}} A_n$$

request $x_n.A_n$ is estimated to be safe along chain H in state S if the predicate $\mathbf{safe}^{\omega}_{S/H}(x_n.A_n)$ holds.

$$\mathbf{safe}^{\omega}_{S/H}(x_n.A_n) = \exists\, i : [\mathcal{ACT}_{S/A_i} \neq \emptyset \bigwedge \mathbf{dep}^{\omega}_{S/H_{i..n}}(x_n.A_n) \subseteq \mathcal{AS}_{S/A_i}]$$

Theorem 6.7

If $\mathbf{safe}^{\omega}_{S/H}(x_n.A_n)$ is true, then all object A_1 dependents of request $x_n.A_n$ along chain H are reflected in the active state of object A_1.

Proof: The proof is by contradiction. Suppose that $\mathbf{safe}^{\omega}_{S/H}(x_n.A_n)$ is true, but that there is an object A_1 request, $x_1.A_1$, that is dependent on request $x_n.A_n$ along chain H

$$x_1.A_1 \prec_{\mathcal{R}} x_2.A_2 \prec_{\mathcal{R}} \cdots \prec_{\mathcal{R}} x_n.A_n$$

but is not reflected in the active state of object A_1.

$$x_1.A_1 \not\in \mathcal{AS}_{S/A_1}$$

Because $\mathbf{safe}^{\omega}_{S/H}(x_n.A_n)$ is true, we know from its definition that there exists some active object, A_i, in the chain such that

$$\mathbf{dep}^{\omega}_{S/H_{i..n}}(x_n.A_n) \subseteq \mathcal{AS}_{S/A_i}$$

From theorem 6.3, we know that $\mathbf{dep}^{\omega}_{S/H_{i..n}}(x_n.A_n)$ contains all of the object A_i dependents of $x_n.A_n$ that occur along chain H, including $x_i.A_i$. It therefore follows that request $x_i.A_i$ is reflected in the active state of object A_i.

$$x_i.A_i \in \mathcal{AS}_{S/A_i}$$

However, request $x_i.A_i$ is also dependent on request $x_1.A_1$. The state of object A_i (an active object) therefore reflects a request ($x_i.A_i$) that is dependent on an object A_1 request ($x_1.A_1$) that is not reflected in that object's active state. The state of the system is therefore observably inconsistent, contradicting the assumption that it is observably consistent. □

The general estimate of the safety of a request, $x.A$, is constructed by combining the estimates of the request's safety along all chains to A from active objects.

Definition 6.16

Let $(\mathcal{R}, \prec_{\mathcal{R}})$ be a request structure, let $\leadsto_{\mathcal{R}}$ be a potential dependency relation consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$, and let S be a system state consistent with $(\mathcal{R}, \prec_{\mathcal{R}})$. A request, $x.A$, is estimated to be safe in state S if the predicate $\mathbf{safe}^{\omega}_{S}(x.A)$ *holds.*

$$\mathbf{safe}^{\omega}_{S}(x.A) = \bigwedge_{\{\, B \in \mathcal{OBJS} \mid \mathcal{ACT}_{S/B} \neq \emptyset \,\}} \; \bigwedge_{H \in \mathcal{BA\text{-}CHAINS}} \mathbf{safe}^{\omega}_{S/H}(x.A)$$

Theorem 6.8

If $\mathbf{safe}^{\omega}_{S}(x.A)$ *holds, then request $x.A$ is safe in state S.*

$$\mathbf{safe}^{\omega}_{S}(x.A) \implies \mathcal{SAFE}_S(x.A)$$

Proof: We show the contrapositive. Suppose that request $x.A$ is unsafe in state S. Then request $x.A$ is dependent on some other request, $y.B$, on an active object ($\mathcal{ACT}_{S/B} \neq \emptyset$) that is not reflected in the object's active state.

$$y.B \notin \mathcal{AS}_{S/B}$$

By definition, this dependency must occur along some chain, H, from object B to object A. From theorem 6.7, the predicate $\mathbf{safe}^{\omega}_{S/H}(x.A)$ must be false. Because this is one of the conjuncts in the definition of $\mathbf{safe}^{\omega}_{S}(x.A)$, the compound safety estimate must also be false. □

6.4 Using the Estimates

Both the basic and compound estimates can be substituted directly into the recovery mechanism as shown below. Because the estimates all have the property that they are sound, they can be used in place of the values of $\overline{\mathcal{CON}}(x.A \prec y.B)$, $\overline{\mathcal{DEP}}_B(x.A)$, and $\overline{\mathcal{SAFE}}_S(x.A)$ without modification of the algorithms.

	$\overline{\mathcal{CON}}(x.A \prec y.B)$	$\overline{\mathcal{DEP}}_B(x.A)$	$\overline{\mathcal{SAFE}}_S(x.A)$
Basic	$\mathbf{con}^{0}_{S}(x.A \prec y.B)$	$\mathbf{dep}^{0}_{S/B}(x.A)$	
Compound	$\mathbf{con}^{\omega}_{S}(x.A \prec y.B)$	$\mathbf{dep}^{\omega}_{S/B}(x.A)$	$\mathbf{safe}^{\omega}_{S}(x.A)$

The compound estimates have the advantage that they are more often defined than the basic estimates. However, the basic estimates are less expensive to compute.

If there is insufficient information in the system to form an estimate required by the recovery mechanism (*i.e.* the estimate is undefined), the mechanism must block and wait for additional servers to recover and provide enough information to construct the estimate. If the undefined estimate occurs in the JOIN phase of recovery, the entire recovery sequence must block. If the undefined estimate occurs in the ACTIVATE phase, then only the activation of the object that required the estimate must block; the recovery mechanism can proceed with the activation of other objects.

6.5 Summary

In this chapter we presented several methods for estimating the dependencies between requests when explicit dependency information is not available in the system. The estimates were divided into two classes: basic estimates and compound estimates. The basic estimates were simple estimates designed to search the orders of requests in servers' logs for evidence of request dependencies. The compound estimates were more complex estimates designed to combine the results of the basic estimates in order to detect transitive dependencies embedded across multiple servers' logs.

Both the basic and compound estimates had the property that they were sound. Because of this, the estimates could be used directly by the log transformations and recovery algorithms. By using sound estimates, the recovery mechanism was guaranteed to ensure all true dependencies between requests, plus possibly a few extraneous orderings. However, because the estimates were sometimes undefined, the recovery mechanism might occasionally need to block and wait until sufficient ordering information is available in the logs of functioning servers to construct the needed estimates.

In order to construct the estimates, we assumed that we were given an approximation of the dependencies between objects, $A \leadsto_{\mathcal{R}} B$, called a potential dependency relation. This relation had the property that it related all objects that had dependent requests. The relation was not required to be precise, however. It could relate objects between which no dependencies existed. However, inaccuracies in a potential dependency relation caused unnecessary restrictions to be placed on the structure of the system. They also caused undefined estimates to occur more often.

Chapter 7

Efficiency Issues

In this chapter we examine several issues regarding the efficiency of the recovery mechanism. We begin by describing a cyclic condition that can arise in the dependency estimates and cause the recovery mechanism to block. By restricting the structure of a system, we show how this cyclic condition can be avoided. We then describe a special class of systems that can be recovered efficiently without blocking using only the basic estimates. Finally, we examine the problem of using checkpoints (of object states) in the recovery mechanism in order to bound the size of logs.

7.1 Cycle Restriction

Even though the dependencies between requests form a partial order, the estimates sometimes generate cyclic orderings. Consider the three logs and potential dependency relation shown below.

$$\begin{array}{|c|}\hline x.A \\ \hline y.B \\ \hline\end{array} \qquad \begin{array}{|c|}\hline y.B \\ \hline z.C \\ \hline\end{array} \qquad \begin{array}{|c|}\hline z.C \\ \hline x.A \\ \hline\end{array} \qquad\qquad \begin{array}{l} A \rightsquigarrow_{\mathcal{R}} B \\ B \rightsquigarrow_{\mathcal{R}} C \\ C \rightsquigarrow_{\mathcal{R}} A \end{array}$$

From this information, the dependency estimates would generate a cyclic ordering for the three requests.

$$x.A \prec y.B \prec z.C \prec x.A$$

At least one of the estimated request dependencies must be spurious. However, based on the information available to the estimates, there is no way of determining which ordering it is.

If a server of objects A, B, and C recovers and attempts to add the three requests to its logs, a problem occurs. Without knowing which request ordering is spurious, any ordering of the three requests within the recovering server's log potentially violates a true dependency. When this situation arises, the recovering server must block and wait until another (failed server's) log becomes available and is able to contradict one of the cyclic orderings.

The problem of estimated cyclic dependencies can be avoided by requiring that any server of an object involved in a potential cycle must also serve all other objects in that cycle. Such a restriction can be easily implemented in a system, such as ISIS [BCJ+], that provides flexibility about which objects a given server manages.

<u>Cycle Restriction</u>

If a cycle exists in the potential dependency relation

$$A_1 \leadsto_{\mathcal{R}} A_2 \leadsto_{\mathcal{R}} \cdots \leadsto_{\mathcal{R}} A_n \leadsto_{\mathcal{R}} A_1$$

then any server that manages one object in the cycle manages all objects in the cycle.

$$\mathcal{SERV}_{A_1} = \mathcal{SERV}_{A_2} = \ldots = \mathcal{SERV}_{A_n}$$

A request, such as $x.A$ above, cannot then be involved in an estimated dependency cycle because any server that logged $x.A$ would also have logged all of its dependents along the cycle ($y.B$ and $z.A$) in some total order within its log, contradicting at least one of the cyclic orderings.

7.2 Backward Inclusion Systems

In general, the compound estimates of chapter 6 are fairly expensive to compute. In order to form a dependency estimate along a particular chain, H, the compound estimates combine approximations constructed along all sub-chains (sub-divisions) of H. Because the number of sub-chains of a chain grows exponentially with the length of the chain, this method can be prohibitively expensive for even modestly sized chains. This cost can be reduced by employing dynamic programming techniques [Den82]. However, for long chains, dynamic programming solutions can also be expensive

Another method for reducing the cost of constructing an estimate is to limit the lengths of the sub-chains considered by the estimation method to a fixed maximum length. This has the effect of reducing the number of sub-chains along which estimates are computed to be polynomial in the length of the chain. Of course, limiting the number of sub-chains considered by the estimation method increases the likelihood that an estimate will be undefined.

In the extreme, we can limit the estimation method to consider only sub-chains of length two; that is, we can limit the recovery mechanism to using only the basic estimates. The basic estimates have the advantage that they are the least expensive estimates to compute, but the disadvantage that they are the most likely estimates to be undefined. However, there is a special class of systems in which the basic estimates are always defined.

Definition 7.1

A system is a <u>*backward inclusion system*</u> *if it satisfies the following condition:*

$$\forall\, A, B \in \mathcal{OBJS} :\ \ A \rightsquigarrow^{*}_{\mathcal{R}} B \implies \mathcal{SERV}_B \subseteq \mathcal{SERV}_A$$

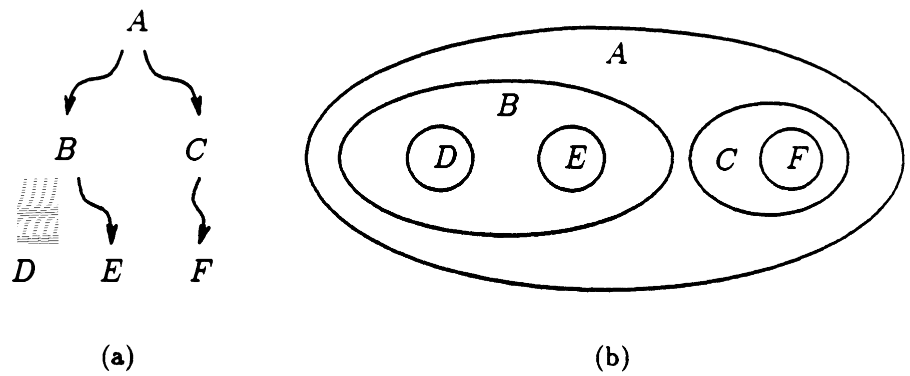


Figure 7.1: A hierarchical backward inclusion system

Intuitively, a system is a backward inclusion system if any server that manages a replica of an object, A, also manages replicas of all objects on which A is potentially dependent. It follows then that if a server logs some request, $x.A$, then it also logs every dependent of $x.A$. Because a request never occurs in a log without all of its dependents, the basic estimates are always defined and the recovery mechanism never aborts. Note that backward inclusion systems satisfy the cycle restriction and so never abort due to cyclic dependency conditions.

The class of backward inclusion systems consists essentially of hierarchically organized systems such as the one depicted in figure 7.1. Figure 7.1(a) shows the potential dependency relation between the six objects in the system and figure 7.1(b) shows the overlap between the server sets of the six objects. The set of backward inclusion systems also includes some non-hierarchical systems such as the one depicted in figure 7.2.

7.3 Chec pinting

As we have preser them, logs grow without bound. In any implementation of the recovery mechanism, the growth of logs must be limited through the use of

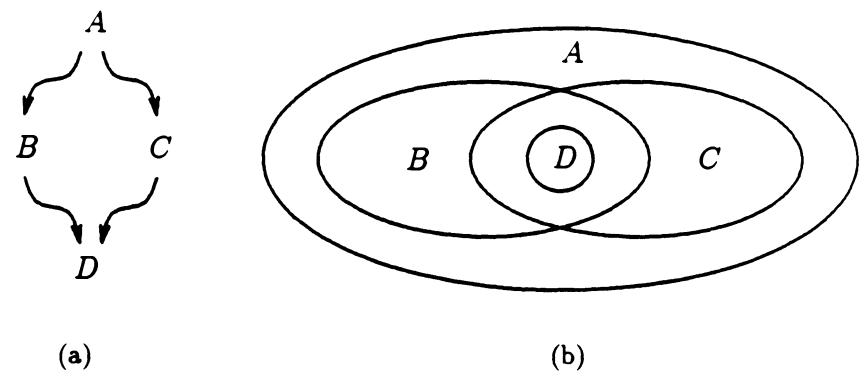


Figure 7.2: A non-hierarchical backward inclusion system

checkpoints. A checkpoint can be logically modeled as a set of requests.

Definition 7.2

The check oint of object A in state S at server f, denoted $\mathcal{CKPT}^{A}_{S/f}$, *is a set of causally consistent requests on object A.*

$$\forall\, x'.A, x.A \in \mathcal{R}\ (x'.A \prec_{\mathcal{R}} x.A):\quad x.A \in \mathcal{CKPT}^{A}_{S/f} \implies x'.A \in \mathcal{CKPT}^{A}_{S/f}$$

In reality, the checkpoint stored by a server is not a set of requests, but a compact representation of the object state corresponding to that set of updates. However, for the purposes of discussion, we choose to model a checkpoint as a set of requests.

A recovering server restores its replica of an object, A, from its log by first restoring the replica to the checkpointed state and then replaying the logged requests on object A. In order to ensure that only consistent states are restored to replicas, the causality condition on logs is extended to include checkpoints. First, the checkpoints and log of a server are restricted to contain only requests on objects managed by the server. Second, if a server logs or checkpoints some request, $x.A$, then it must previously have logged or checkpointed all dependents of $x.A$ (on objects managed by the server). Because checkpoints precede all other entries in a log, this implies that a server that has checkpointed $x.A$ has also checkpointed the dependents of $x.A$. Lastly, the checkpoints and log of a server are restricted from containing any duplicate requests.

Definition 7.3

The log, $(\mathcal{L}_{S/f}, \rightarrow_{S/f})$, of a server f in state S is consistent with a request structure, $(\mathcal{R}, \prec_{\mathcal{R}})$, if

1. $\forall\ x.A \in \mathcal{L}_{S/f} :\ A \in \mathcal{OBJS}_f$

   $\forall\ A \in \mathcal{OBJS}_f :\ \ \mathcal{CKPT}^A_{S/f}$ *contains only object A requests*

2. $\forall\ x.A \in \mathcal{L}_{S/f} :\ \forall\ y.B \in \mathcal{R}\ (y.B \prec_{\mathcal{R}} x.A) :$

   $B \in \mathcal{OBJS}_f \implies$

   $[\ y.B \in \mathcal{CKPT}^B_{S/f} \vee (y.B \in \mathcal{L}_{S/f} \wedge y.B \rightarrow_{S/f} x.A)\ ]$

3. $\forall\ A, B \in \mathcal{OBJS}_f :$

   $\forall\ x.A \in \mathcal{CKPT}^A_{S/f} :$

   $\forall\ y.B \in \mathcal{R}\ (y.B \prec_{\mathcal{R}} x.A) :\ y.B \in \mathcal{CKPT}^B_{S/f}$

4. $\forall\ A \in \mathcal{OBJS}_f :\ \ \mathcal{CKPT}^A_{S/f} \cap \mathcal{L}_{S/f}\ = \emptyset$

The projection operator is also extended to account for checkpoints in the following way:

Definition 7.4

The projection of a log, $(\mathcal{L}_{S/f}, \rightarrow_{S/f})$, onto an object, $A \in \mathcal{OBJS}$, is

$$(\mathcal{L}_{S/f}, \rightarrow_{S/f})\ |_A\ = \{\ x.A \mid x.A \in \mathcal{L}_{S/f} \bigvee x.A \in \mathcal{CKPT}^A_{S/f}\ \}$$

The main difficulty involved in implementing checkpoints is ensuring that the causal consistency restrictions are not violated. For example, the log addition transformation must be careful not to add to a server's log any request that is already present in that server's checkpoints. Similarly, a checkpoint should never be installed at a server if that checkpoint reflects a request already present in the server's log (this can be a problem when a new checkpoint is transferred to a recovering server during the server's JOIN phase).

These problems can be solved by storing, with each checkpoint, explicit information about the requests it reflects. Duplicates can then be detected and removed from the affected log. Due to the large number of requests that may be reflected in a checkpoint, however, it will generally be impractical to maintain such explicit information.

Another method for avoiding duplicates is to use implicit information contained in other servers' logs. For example, if a server, f, known to be consistent, has logged some request, $x.A$, then the checkpoint of object A at server f cannot reflect $x.A$. It therefore follows that request $x.A$ can be added to the log of any server, with the same object A checkpoint as f, without introducing a duplicate into its log. By adapting a checkpointing algorithm such as [KT87], we can increase the likelihood that servers will have identical checkpoints.

7.4 Summary

In this chapter we examined several issues concerning the efficiency of the recovery mechanism. We began by describing a circularity condition that can arise in the estimates and cause the recovery mechanism to abort. We showed how this problem could be avoided by restricting the structure of the system. We then outlined a special class of systems, called *backward inclusion systems*, that were efficiently solvable without blocking using the basic estimates. Finally, we outlined some of the problems involved in adding object checkpoints to server logs.

Chapter 8

Grouping Consistency

This dissertation has presented a recovery mechanism for preserving causal consistency in a distributed system. The basic principles of estimating dependencies between requests and using those estimates to preserve consistency can also be applied to other forms of consistency. In this chapter we outline changes in the recovery mechanism for supporting an atomic form of consistency called *grouping consistency*.

8.1 Grouping Consistency

Under grouping consistency, requests may be collected into sets (called *groups*) with the property that no request in a group is reflected in the system unless all of the requests in the group are also reflected. The requests in a group do not have any ordering properties between them, only the all-or-none property. Grouping consistency differs from serializability in that there are no ordering properties between the requests in different groups; they may be received and processed by servers in any order.

As an example of grouping consistency, consider an airline reservation system. Suppose that a passenger wishes to make a reservation on a pair of connecting flights. This operation can be implemented as two separate requests. First, a seat

Request Structure: $(\mathcal{R}, =_{\mathcal{R}})$

$$\mathcal{R} = \{res_1.A,\ res_2.B,\ res_3.A,\ res_4.B\}$$

$$res_1.A =_{\mathcal{R}} res_2.B \qquad res_3.A =_{\mathcal{R}} res_4.B$$

Figure 8.1: A grouping request structure

is reserved for the passenger on the first flight, A. Second, a seat is reserved for the passenger on the connecting flight, B. In order to be consistent, the system should never reflect one seat reservation without reflecting the other. The two reservations would therefore be collected into a group and submitted as a unit.

We can modify the definition of a request structure to reflect groupings of requests in the following way.

Definition 8.1

A request structure, $(\mathcal{R}, =_{\mathcal{R}})$, is a set of requests along with an equivalence relation on that set.

Here, $\mathcal{R}$ is the set of client requests and $=_{\mathcal{R}}$ relates all grouped requests. If two requests are related, $x.A =_{\mathcal{R}} y.B$, then the system must reflect both requests or neither request. Note that a request may belong to multiple groups. If request $x.A$ is grouped with request $y.B$ ($x.A =_{\mathcal{R}} y.B$), and request $y.B$ is separately grouped with request $z.C$ ($y.B =_{\mathcal{R}} z.C$), then by the transitivity of the grouping relation request $x.A$ cannot be reflected in the system unless request $z.C$ is also reflected.

Figure 8.1 shows a request structure for the airline reservation system described above. The system consists of four seat reservations ($res_1.A$, $res_2.B$, $res_3.A$, and $res_4.B$) on two separate flights (A and B). In the example, $res_2.B$ is a connecting reservation from $res_1.A$ and $res_4.B$ is a connecting reservation from $res_3.A$.

We assume servers receive, process, and log grouped requests as a unit. As a result, server logs are consistent with the group structure on requests. That is, if the log of a server reflects some request, $x.A$, then it also reflects all requests related to $x.A$ (on objects managed by the server).

Definition 8.2

The log, $(\mathcal{L}_{S/f}, \rightarrow_{S/f})$, of server f in state S is consistent with a request structure, $(\mathcal{R}, =_{\mathcal{R}})$, if

1. $\forall\ x.A \in \mathcal{L}_{S/f} :\ f \in \mathcal{SERV}_A$
2. $\forall\ x.A \in \mathcal{L}_{S/f} :$
$$\forall\ y.B \in \mathcal{R}\ (x.A =_{\mathcal{R}} y.B) :\quad f \in \mathcal{SERV}_B \implies y.B \in \mathcal{L}_{S/f}$$

As before, we assume that servers recover in observably consistent states. That is, at the time of a server recovery, the logs of all functioning servers are consistent with the application's request structure and all active servers of an object reflect the same object state. Further, the states of different active objects are mutually consistent: if a request is reflected in the active state of one object, then all of its dependents (on active objects) are reflected in their object's active states.

Definition 8.3

A system state, S, is observably consistent with a request structure, $(\mathcal{R}, =_{\mathcal{R}})$, if

1. $\forall\ f \in \mathcal{SERV} - \mathcal{FAIL}_S\ :\ (\mathcal{L}_{S/f}, \rightarrow_{S/f})$ *is consistent with* $(\mathcal{R}, =_{\mathcal{R}})$.
2. $\forall\ A \in \mathcal{OBJS} :\ \forall\ f, g \in \mathcal{ACT}_{S/A} :\ (\mathcal{L}_{S/f}, \rightarrow_{S/f})\ |_A\ =\ (\mathcal{L}_{S/g}, \rightarrow_{S/g})\ |_A$
3. $\forall\ A, B \in \mathcal{OBJS}\ (\mathcal{ACT}_{S/A} \neq \emptyset\ \wedge\ \mathcal{ACT}_{S/B} \neq \emptyset) :$
$$\forall\ x.A \in \mathcal{AS}_{S/A} :\ \forall\ y.B \in \mathcal{R}\ (x.A =_{\mathcal{R}} y.B) :\ y.B \in \mathcal{AS}_{S/B}$$

8.2 Changes to Recovery Mechanism

Recovery under grouping consistency is handled in the same manner as it was under causal consistency. The recovery sequence of a server is divided into two phases. During the JOIN phase, a recovering server receives and installs the current states of active objects. During the ACTIVATE phase, a recovering server constructs and installs new (consistent) states for inactive objects.

The algorithms implementing the JOIN and ACTIVATE phases are nearly identical to those of chapter 5. However, the log transformations on which they are built must be modified to account for the new consistency definition. Consider the log addition transformation. When a request is added to a server's log, the transformation must be certain that all requests (directly or transitively) grouped with it are also present in the log. If they are not, then the transformation must add them.

Definition 8.4

The set of object B dependents of request $x.A$ under grouping consistency are

$$\mathcal{DEP}_B(x.A) \;=\; \{y.B \in \mathcal{R} \mid y.B =_{\mathcal{R}} x.A\}$$

Figure 8.2 shows the complete log addition transformation under grouping consistency. Note that the transformation places no particular ordering on the requests in the log because requests are not ordered under grouping consistency.

The deletion transformation is modified in a similar manner. When a request is deleted from a log, all requests grouped with it are also deleted. The complete log deletion transformation is shown in figure 8.3. Note that although the transformation preserves the order of requests that remain in the log, this restriction is unnecessary.

$$\mathbf{add}_Q(\mathcal{L}_f, \rightarrow_f) \;=\; (\mathcal{L}, \rightarrow_\mathcal{L})$$

where

$$\mathcal{L} \;=\; \mathcal{L}_f \bigcup Q \bigcup \Big[\bigcup_{x.A \in Q} \; \bigcup_{B \in \mathcal{OBJS}_f} \mathcal{DEP}_B(x.A) \;\Big]$$

$\rightarrow_\mathcal{L}$ is any ordering of the requests.

Figure 8.2: Log addition under grouping consistency

$$\mathbf{delete}_Q(\mathcal{L}_f, \rightarrow_f) \;=\; (\mathcal{L}, \rightarrow_\mathcal{L})$$

where

$$\mathcal{L} \;=\; \{\; x.A \in \mathcal{L}_f \mid x.A \notin Q \;\wedge\; \nexists\, y.B \in Q :\; y.B =_\mathcal{R} x.A \;\}$$

$$\forall\, x.A, y.B \in \mathcal{L} :\; (x.A \rightarrow_\mathcal{L} y.B) \Leftrightarrow (x.A \rightarrow_f y.B)$$

Figure 8.3: Log deletion under grouping consistency

When explicit dependency information is not available to the transformations, dependency estimates can be used to preserve consistency. The changes necessary to use estimates in the log transformations are left to the reader.

8.3 Estimating Dependencies

Our estimates of request groupings are divided into two classes: basic and compound. As before, the compound estimates are more accurate and more often defined than the basic estimates, but are also more expensive to compute. However, all estimates have the property that they do not under-estimate the true set of grouped requests. That is, all of the estimates are sound.

We assume that the estimates have access to a *potential dependency relation* that relates pairs of potentially dependent objects. Like the potential dependency relation under causal consistency, this relation should not under-estimate the true set of related objects.

Definition 8.5

A potential dependency relation, $\approx_{\mathcal{R}}$, over request structure $(\mathcal{R}, =_{\mathcal{R}})$, is a binary relation on the objects in $\mathcal{OBJS}$ with the property that it relates all pairs of objects between which dependencies hold.

$$\forall\ x.A, y.B \in \mathcal{R}: \quad x.A =_{\mathcal{R}} y.B \implies A \approx_{\mathcal{R}} B$$

8.3.1 Basic Estimates

The basic estimates are designed to search individual server logs for evidence of request groupings. We begin by presenting an estimate of when two requests are *not* grouped. This estimate is then used to construct an estimate of the complete set of (grouped) dependents of a request.

Consider the problem of estimating when two requests, $x.A$ and $y.B$, are *not* grouped. Because server logs are consistent with the request structure of an application, we know that the requests are not grouped if a server of objects A and B has logged one request, but not the other. Because the states of active objects are consistent with the application's request structure, we also know that $x.A$ and $y.B$ are not grouped if both objects are active, but only one of the requests is reflected in its object's active state. Combining these observations with the knowledge provided by the potential dependency relation we derive the following estimate.

Definition 8.6

Let $(\mathcal{R}, =_{\mathcal{R}})$ be a request structure, let $\approx_{\mathcal{R}}$ be a potential dependency relation consistent with $(\mathcal{R}, =_{\mathcal{R}})$, and let S be a system state consistent with $(\mathcal{R}, =_{\mathcal{R}})$. The request grouping, $x.A = y.B$, is directly contradicted in state S, denoted $\mathbf{con}^0_S(x.A = y.B)$*, if any of the following three conditions holds:*

1. $A \not\approx_{\mathcal{R}} B$
2. $\exists\ f \in \mathcal{FUNC}_{S/A} \cap \mathcal{FUNC}_{S/B}$:
   $[(x.A \in \mathcal{L}_{S/f} \ \wedge\ y.B \notin \mathcal{L}_{S/f}) \vee (y.B \in \mathcal{L}_{S/f} \ \wedge\ x.A \notin \mathcal{L}_{S/f})]$
3. $\mathcal{ACT}_{S/A} \neq \emptyset \ \wedge\ \mathcal{ACT}_{S/B} \neq \emptyset \ \wedge$
   $[(x.A \in \mathcal{AS}_{S/A} \ \wedge\ y.B \notin \mathcal{AS}_{S/B}) \vee (y.B \in \mathcal{AS}_{S/B} \ \wedge\ x.A \notin \mathcal{AS}_{S/A})]$

Now consider the problem of estimating the complete set of object B requests grouped with request $x.A$. If a server of objects A and B has logged request $x.A$, then its log must also contains all of the object B dependents of $x.A$. The set of object B requests in its log can therefore be used as an estimate of the dependency set. Additionally, if objects A and B are both active, and the state of A reflects request $x.A$, then the state of B must reflect all of the dependents. The set of requests reflected in the state of B can therefore also be used as an estimate of the dependency set. Combining these approximations along with the information in the preceding estimate, we derive the following estimate of $\mathcal{DEP}_B(x.A)$.

Definition 8.7

Let $(\mathcal{R}, =_{\mathcal{R}})$ be a request structure, let $\approx_{\mathcal{R}}$ be a potential dependency relation consistent with $(\mathcal{R}, =_{\mathcal{R}})$, and let S be a system state observably consistent with $(\mathcal{R}, =_{\mathcal{R}})$. For any object $B \in \mathcal{OBJS}$ and request $x.A \in \mathcal{R}$, the <u>basic estimated dependents</u> of $x.A$ are:

$$\mathbf{dep}^0_{S/B}(x.A) = \begin{cases} \perp & \text{if } \neg\exists f \in \mathcal{FUNC}_{S/A} \cap \mathcal{FUNC}_{S/B} :\ x.A \in \mathcal{L}_{S/f} \\ & \qquad\qquad\qquad\qquad \text{and} \\ & \quad \mathcal{ACT}_{S/A} = \emptyset \ \vee\ \mathcal{ACT}_{S/B} = \emptyset \ \vee\ x.A \notin \mathcal{AS}_{S/A} \\ \emptyset & \text{if } B \not\approx_{\mathcal{R}} A \\ \{y.B \mid \neg\mathbf{con}^0_S(y.B = x.A) \ \wedge & \qquad\qquad\qquad\qquad \text{o.w.} \\ \quad [\ \exists f \in \mathcal{FUNC}_{S/A} \cap \mathcal{FUNC}_{S/B} : x.A, y.B \in \mathcal{L}_{S/f} & \\ \quad\ \vee\ y.B \in \mathcal{AS}_{S/B}\ ]\ \} & \end{cases}$$

8.3.2 Compound Estimates

The information necessary to detect a request grouping may be distributed across multiple logs. For example, suppose that there is a grouping between n different requests.

$$x_1.A_1 \ =_{\mathcal{R}}\ x_2.A_2 \ =_{\mathcal{R}}\ \cdots\ =_{\mathcal{R}}\ x_n.A_n$$

This grouping may embed itself across $n-1$ logs in the following way.

$x_1.A_1$	$x_2.A_2$	$\cdots$	$x_{n-1}.A_{n-1}$
$x_2.A_2$	$x_3.A_3$		$x_n.A_n$

Using the basic estimates, we would detect each of the individual grouping pairs:

$$x_1.A_1 =_{\mathcal{R}} x_2.A_2 \qquad x_2.A_2 =_{\mathcal{R}} x_3.A_3 \qquad \ldots \qquad x_{n-1}.A_{n-1} =_{\mathcal{R}} x_n.A_n$$

In order to detect the overall grouping between the n requests, the results of the basic estimates must be combined. This can be done using the compound estimates of chapter 6. By substituting the preceding basic estimates for those of chapter 6, the compound estimates will approximate request groupings instead of causal dependencies. No other modifications are required to the compound estimates.

8.4 Summary

This chapter outlined modifications to the recovery mechanism for supporting a new form of consistency called grouping consistency. Under grouping consistency, requests were collected into sets with the property that no request in a set was reflected in the system unless all requests in the set were reflected.

The recovery sequence of a server remained the same as it was under causal consistency. During the JOIN phase, a recovering server restored its replicas of active objects to those objects' current states. During the ACTIVATE phase, a recovering server restored its replicas of inactive objects to states consistent with the rest of the system. However, the log transformations out of which the recovery algorithms are built had to be modified to account for the new consistency definition.

When explicit information about the groupings of requests was unavailable, the log transformations could use estimates of the groupings in order to preserve consistency in the system. These estimates were divided into two classes: basic and compound. The compound estimates remained the same as they were in chapter 6. However, the basic estimates out of which they are built were redefined to approximate grouping dependencies instead of causal dependencies.

Chapter 9

Conclusions

This dissertation has presented a recovery mechanism for restoring casually consistent states to replicated data objects. The mechanism was based on maintaining logs of the updates that occur to objects, and using those logs to reconstruct object states after failures. Unlike existing techniques, our method does not require any explicit information about the dependencies between updates. Instead. any necessary information about the ordering between requests is inferred from their orderings within logs.

Without a recovery mechanism, two types of inconsistencies develop in a system. First, inconsistencies develop between the different replicas of an object. When a server of a replica recovers from a failure, its log reflects the state of the object from the time of the failure. If the state of the object has changed since the failure, the server will restore an outdated state to its replica. Second, inconsistencies develop between the states of different objects. When all servers of an object fail, some updates on the object may be lost. The state later recovered by the servers may then be missing some requests on which other active objects depend.

Based on these two types of inconsistencies, the recovery sequence of a server is divided into two phases. During the JOIN phase, a recovering server restores its replicas of active objects. The current states of these objects are transferred to the server and written to its log. During the ACTIVATE phase, a server restores its replicas of inactive objects. All recovering servers of an inactive object cooperate in choosing a new state for the object that is consistent with the states of the other objects in the system. Once chosen, the servers modify their logs to reflect this new state.

The algorithms implementing the JOIN and ACTIVATE phases are relatively straight forward. The only difficulty involves preserving the consistency of a server's log when modifications are made to it. The log addition transformation ensures that no request is added to a server's log without all of its dependents. The log deletion transformation ensures that no request is deleted from a log without also removing all requests that depend on it.

When explicit information about request dependencies is not available, the recovery algorithms (as well as the log transformations out of which they are built) can use estimates of the dependencies. In order to preserve consistency in the system, these estimates must have the property that they do not under-estimate the orderings between requests. We presented several dependency estimates with this property. The basic estimates are simple approximations based on searching server logs for evidence of request orderings. The compound estimates are more complicated approximations formed by combining the results of the basic estimates. Although the compound estimates are more accurate and more often defined than the basic estimates, they are also more expensive to compute. We showed that in a special class of systems (the backward inclusions systems) the inexpensive basic estimates can always be used without the possibility of blocking.

Our basic recovery approach can also be applied to forms of consistency other than casual consistency. We showed that with little modification, our recovery technique could be applied to an atomic form of consistency called grouping consistency. Particularly interesting was the fact that the compound estimates remained unchanged between causal and grouping consistency. Only the basic estimates needed to be changed to allow for the new consistency definition.

9.1 Future Work

We conclude this dissertation by discussing several related areas for future research.

9.1.1 Implementation Considerations

A recovery mechanism based on the ideas in this dissertation was implemented in the ISIS system [BCJ+]. In ISIS, the server set of an object is implemented as a *process group*. Each process in a group is equivalent to one server and manages one replica of the object. Process groups in ISIS are given unique names. Updates on an object can be broadcast to the group using only the group name. When such a broadcast occurs, ISIS automatically resolves the name of the group into its current set of member processes and delivers a copy of the update broadcast to each member.

Unfortunately, the exact recovery mechanism described in this dissertation could not be implemented in ISIS because of the way in which ISIS handles process groups. When a process (server) recovers in ISIS, it is required to re-join the process groups (object server sets) that it previously belonged to in a fixed order that is set at the time the application is written. However, the recover sequence presented in chapter 3 requires a recovering server to join object groups in flexible orders. When a server recovers, it must first JOIN the server sets of all objects that are currently active (whatever they are) and then ACTIVATE its replicas of objects that are inactive. We believe that ISIS could be made to support processes joining process groups in flexible orders. However, the modifications would require substantial revision of the code, and our current applications that require such support.

Like the recovery mechanism described in this dissertation, the recovery mechanism in ISIS automatically ensures consistency between the replicas of an object. However, the ISIS recovery mechanism does not provide automatic consistency between the states of different object. Instead, it ensures that the state of an inactive object is always recovered using the log of the last server of the object to fail [Ske85]. By allowing clients to force certain updates to be logged by all functioning servers of an object, clients can control which updates may be lost from the system, and therefore control consistency in the system.

Beyond the ability to join process groups in flexible orders, ISIS should provide a good platform on which to build the recovery mechanism described in this dissertation. ISIS currently supports a state transfer mechanism whereby a server (process) joining or re-joining an active object server set (process group) is automatically transferred the current state of the object (process group). This state transfer appears atomic from the point of view of a client, so each update broadcast to the object (process group) is processed by all of its members in the same state of the object (process group). This state transfer mechanism is used by the current ISIS recovery mechanism to initialize replicas of active objects at recovering servers.

The ISIS broadcast mechanism also provides a facility for automatically collecting replies to message broadcasts, including the handling of failures during the broadcast-reply sequence. This facility should prove invaluable in the dissemination and collection of basic dependency information. For example, a recovering process requiring dependency information about certain updates could broadcast a request to the servers of the objects involved. Upon receiving the request, the servers could reply with the current states of the objects and ordering information from their logs. Using simple unions and intersections, the recovering process could then combine this information to form the necessary estimates. This type of mechanism would be sufficient for building backward inclusion systems, where only basic dependency information is required.

This technique could also be used to compute the compound estimates. However, doing so would be costly, not only in terms of time, but also in terms of space and message traffic. In order to form the compound estimates needed for recovery, a server must collect basic estimates from the logs of many different servers. This collection process can potentially create a large load of message traffic at the recovering server. Further, once the basic estimates are collected, the server must combine them to form the compound estimates. If the potential dependency relation contains long chains, this could require significant time and space.

In order to reduce the time, space, and message load at a recovering server, the task of computing estimates could be distributed across the functioning servers in the system. Each functioning server could locally compute the basic estimates related to the objects it manages. This would introduce only a limited amount of message traffic at each server. Once the basic estimates are computed, the functioning servers could exchange their results and combine them in a hierarchical fashion in order to form the overall compound estimates.

9.1.2 Other Consistency Forms

We have described variants of our recovery mechanism for implementing both causal consistency and grouping consistency. An interesting problem is whether these variants can be combined to implement serializable consistency. Grouping consistency provides the all-or-none property required by serializability. Causal consistency might then be added to implement some type of ordering between the requests in different groups.

A related problem concerns the types of consistency that can be enforced using our basic mechanism. We would like to characterize the forms of consistency

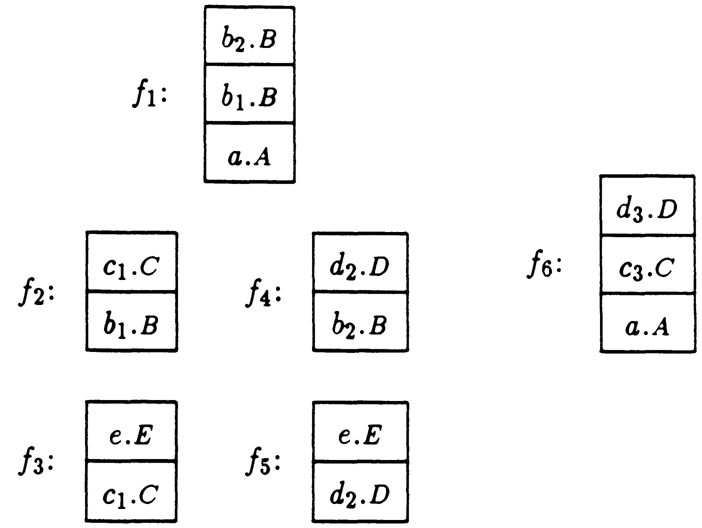


Figure 9.1: Logs generating non-optimal estimates

implementable using dependency estimates. The compound estimates of chapter 6 apply equally well to both causal and grouping consistency. The question then naturally arises as to whether these estimates apply to more general forms or classes of consistency.

9.1.3 Optimal Estimates

The compound estimates of chapter 6 are not optimal in the sense that they may occasionally yield an ordering between two requests, even when there is evidence available in the system to contradict the ordering. For example, consider the set of logs shown in figure 9.1. This figure depicts the logs of six servers (f_1, f_2, f_3, f_4, f_5, and f_6), each server managing only those objects for which requests are shown in its log. Suppose that the potential dependency relation in this system forms one long chain.

$$E \leadsto_{\mathcal{R}} D \leadsto_{\mathcal{R}} C \leadsto_{\mathcal{R}} B \leadsto_{\mathcal{R}} A$$

Applying the compound estimates to these logs, the estimates would yield an ordering between requests $a.A$ and $e.E$.

$$e.A \prec a.A$$

However, from the logs we can determine that this ordering is not possible. Any dependency of request $a.A$ on request $e.E$ must occur along the chain of objects depicted above (in the potential dependency relation). From the log of server f_1, we know that any such dependency would include either request $b_1.B$ or $b_2.B$. If the dependency included request $b_1.B$, then from the log of server f_2 we know that it must also include request $c_1.C$. This implies that $a.A$ is dependent on $c_1.C$. But, this ordering is contradicted by the log of server f_6. Similarly, if the dependency chain includes request $b_2.B$, then from the log of server f_4 we know that is also includes $d_2.D$. This implies that request $a.A$ is dependent on request $d_2.D$. But, this ordering is also contradicted by the log of server f_6.

An interesting problem would be to determine an optimal set of dependency estimates that yield an efficient implementation. As we pointed out earlier, the compound estimates apply equally well to both causal and grouping consistency. We would like to find an optimal set of estimates that also have this property, preferably extending to other consistency forms as well. Because it has not been the goal of this dissertation to pursue complexity issues, we will not make any general speculations about the difficulty of computing an optimal set of estimates. We would like to point out, however, that the problem of determining an optimal set of estimates is reminiscent of other optimality results in the literature that have been shown to be NP-complete [Pap79].

Bibliography

[AM83] J. E. Allchin and M. S. McKendry. Synchronization and recovery of actions. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 31–44. ACM, August 1983.

[BCJ+] Kenneth P. Birman, Robert Cooper, Thomas A. Joseph, Kenneth P. Kane, and Frank Schmuck. *ISIS - A Distributed Programming Environment: User's Guide and Reference Manual.* The ISIS Project, Department of Computer Science, Cornell University, Ithaca, New York 14853.

[BG81] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 12(2):185–221, June 1981.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, first edition, 1987.

[BJ87a] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 123–138. ACM, November 1987.

[BJ87b] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[CASD86] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. Research Report RJ 5244 (54244), IBM, July 1986.

[CM84] J. M. Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Coo85] Eric Cooper. Replicated distributed programs. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 63–78. ACM, December 1985.

[CP86] Douglas E. Comer and Larry L. Peterson. Conversation-based mail. *ACM Transactions on Computer Systems*, 4(4):299–319, November 1986.

[Den82] Eric V. Denardo. *Dynamic Programming: Models and Applications.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, first edition, 1982.

[DGMS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[FC87] Ross S. Finlayson and David R. Cheriton. Log files: An extended file service exploiting write-once storage. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 139–148. ACM, November 1987.

[Gra78] J. Gray. Notes on database operating systems. In *Lecture Notes in Computer Science 60.* Springer-Verlag, Berlin, 1978.

[HMSC88] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery management in quicksilver. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988.

[J+87] David R. Jefferson et al. Distributed simulation and the time warp operating system. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 77–93. ACM, November 1987.

[Jef85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[JLHB87] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 105–106. ACM, November 1987.

[JZ87] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *The Seventeenth International Symposium on Fault-Tolerant Computing*, pages 14–19. IEEE, July 1987.

[JZ88] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181. ACM, August 1988.

[KT87] Richard Koo and Sam Toueg. Checkpointing and rollback recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, January 1987.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LCJS87] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of argus. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 111–122. ACM, November 1987.

[LL86] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 29–39. ACM, August 1986.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[OLS85] Brian M. Oki, Barbara H. Liskov, and Robert W. Scheifler. Reliable object storage to support atomic actions. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 147–159. ACM, December 1985.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[PBS89] Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[PP83] Michael L. Powell and David L. Presotto. Publishing: a reliable broadcast communication mechanism. In *Proceedings of the Nineth ACM Symposium on Operating System Principles*, pages 100–109. ACM, October 1983.

[PT86] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, SE-12(3):477–482, March 1986.

[Sch88] Frank Bernhard Schmuck. *The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems.* Ph.D. dissertation, Cornell University, August 1988.

[Ske85] Dale Skeen. Determining the last process to fail. *ACM Transactions on Computer Systems*, 3(1):15–30, February 1985.

[SS83] R. Schlichting and F. Schneider. Fail-stop processors: An approach to designing fault-tolerant distributed computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[SY85] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[Ull82] Jeffrey D. Ullman. *Principles of Database Systems*, chapter 11. Computer Science Press, 11 Taft Court, Rockville, Maryland 20850, second edition, 1982.