Causally Consistent Recovery of Partially Replicated Logs

Kenneth P. Kane
Kenneth P. Birman*

88-949
November 1988

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

*This work was supported by DARPA (DoD) under ARPA order 6037, Contract N00140-87-C-8904, also by a grant from the Siemens Corporation. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy or decision.

Causally Consistent Recovery of Partially Replicated Logs

Kenneth P. Kane and Kenneth P. Birman

Cornell University
Computer Science Department
Ithaca, New York 14853 USA

November 28, 1988

Abstract

An algorithm is presented for the consistent recovery of replicated data in a client-server system. The algorithm is based on logging and is similar to the optimistic techniques that are well known in the literature. However, unlike in existing optimistic techniques, explicit dependency information is not maintained. Instead, dependency information is estimated from the ordering of messages found in servers' logs. These dependency estimates can, in general, be expensive to compute. It is therefore shown how inexpensive estimates can be applied when a system is well structured.

1 Introduction

Object oriented distributed systems are becoming increasingly common. These systems provide users with tools for building abstract data objects. Such an object generally consists of routines for maintaining it along with an interface by which clients access it. Only the interface of an object is visible to a client; implementation details, such as replication and failure

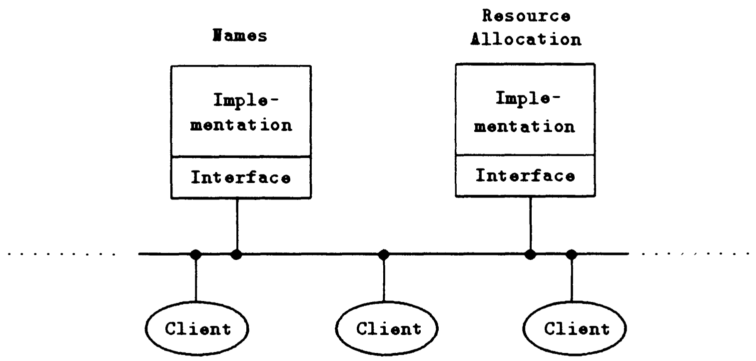

Figure 1: A portion of a distributed operating system. Depicted are two objects representing a name server and a resource allocation manager. Clients or processes in the system operate by first registering themselves with the naming service and then allocating resources under that name.

recovery, are hidden within the object module. Figure 1 depicts such a system.

Failure recovery in these systems is often accomplished through the use of *logging*. By writing to a log file the sequence of updates that occurs to an object, the object's state can be reconstructed after a failure. However, because the states of objects may be related, consistency problems potentially arise if object logs are not coordinated. For example, in the system of figure 1 the state of the resource manager is dependent on the state of the name server; only registered clients may allocate resources. Suppose a failure causes a client registration record to be lost (not logged). If resource allocations *are* logged for this client, then the system may later recover into a state that reflects the client's allocations without reflecting its registration.

Transactions can be used to enforce consistency between logs. For example, the registration of a client name and its allocation of resources could be grouped into a single transaction and committed as a unit, in order to ensure that they are recovered atomically. However, many applications do not require the full power of atomicity that transactions provide. Often a weaker form of consistency, such as causal consistency, is sufficient to guarantee correctness [Lam78,BJ87a]. In loosely coupled systems such as ISIS [BJ87a], this weakening of consistency usually leads to improved performance and availability.

This paper presents a log-based mechanism for the causally consistent recovery of replicated data objects. The problem of representing and maintaining causal dependency information about the updates on objects is not a simple one. Solutions to this problem have been devised for many different settings, including inter-process communication [BJ87b,PBS88], highly available distributed services [LL86], and optimistic failure recovery [SY85,JZ88]. Dependency information in these systems is maintained explicitly: each object update is tagged with either an enumeration of the updates on which it is dependent or with a timestamp that reflects the update's causal ordering.

Unfortunately, it can be difficult or impossible to maintain explicit dependency information about updates when the set of object clients is either unknown or large and dynamically changing. The recovery algorithm presented in this paper avoids the need to maintain explicit dependency information by estimating such information from the ordering of updates in object servers' logs. When an object server first recovers from a failure, it approximates the set of dependencies in the system from ordering information available in the logs of servers. This information is then used to ensure that only consistent object states are recovered.

The presentation of the algorithm is divided into two parts. First, in sections 2 through 6, a recovery algorithm is derived based on explicit knowledge of the dependencies between object updates. In section 2, the formal system model is presented and in section 3 the notions of consistency and correctness are defined. Based on these definitions, section 4 outlines several consistency problems that arise through the use of logging and presents a basic sketch of the recovery mechanism. The actual implementation of the recovery algorithm is built on functions for consistently adding and deleting entries from logs. These functions are presented in section 5 and used in section 6 to describe the recovery algorithm.

The second part of the presentation discusses methods for estimating dependency information from the ordering of updates in object logs. Section 7 presents several dependency estimates and describes how they can be used in the recovery algorithm in place of the values they approximate. In general, the estimates used can be expensive to compute. Because of this, section 8 describes a special class of systems in which inexpensive estimates can be used by the algorithm.

The material presented in this paper is a summary of that in [Kan89]. Much of the formalism and all of the proofs have been omitted for the purposes of brevity.

2 System Model

2.1 Partial Replication

Our system model is a *partially replicated* variation of the client-server model of computation. A set of servers, denoted $SERV$, are used to maintain replicas of a set of data objects. Each server maintains replicas of several different objects. Data objects are not fully replicated: each object is replicated at only some of the servers. We let D denote the set of all data objects in the system and let $SERV_A$ denote the subset of servers managing a replica of object A ($A \in D$).

Objects are accessed by a set of clients that may or may not differ from the set of servers. In order to access an object, $A \in D$, a client broadcasts a request to all servers of A, that is, to all members of $SERV_A$. Upon receiving such a request, each server of object A performs the requested operation on its local copy of A.

We make no assumption about the relative ordering of client requests.

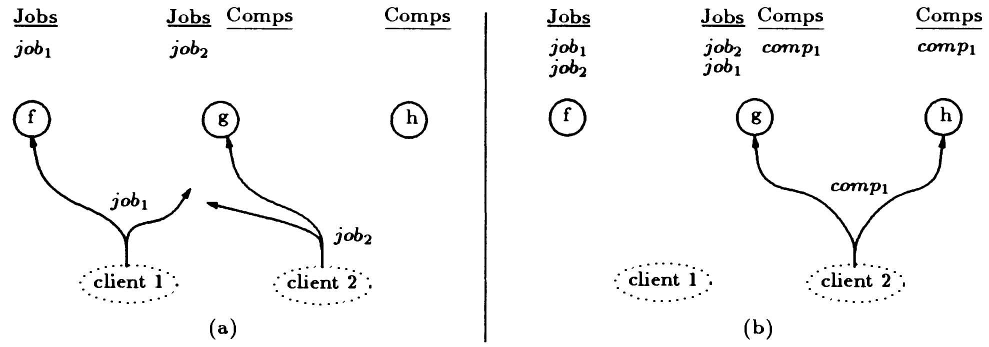

Figure 2: An example of a partially replicated printer service. In figure 2(a), both system clients are broadcasting job submissions. In figure 2(b), the job submissions have completed with the second client broadcasting a notification of the completion of the first job.

Servers may receive the same requests in differing orders, if the orders are mutually consistent and correct with respect to the application being implemented. It is the responsibility of clients to ensure that such correct orderings are perceived by the servers. To this end, clients may use a variety of broadcast mechanisms, each differing in the ordering properties it provides.

As an example, consider figure 2. This figure depicts two states in the execution of a system maintaining information about a printer service. The system consists of three servers (f, g, and h) replicating two data objects (*Jobs* and *Comps*). The object *Jobs* is a list of jobs that have been submitted for printing and is replicated at servers f and g. The object *Comps* is a list of completed jobs and is replicated at servers g and h. Figure 2(a) depicts a state of the system in which two clients (*client* 1 and *client* 2) are submitting jobs for printing. Note that the job submissions will be received in different orders by the two servers of object

Jobs. Figure 2(b) depicts a later state of the system after which both job submission broadcasts (job_1 and job_2) have completed. In this state, *client* 2 is in the process of broadcasting a completion notification for job_1.

2.2 Logging

In order to support recovery from failures, each server maintains a log of the requests that it receives.

Definition 2.1 A *log* is a totally ordered set $(L, \rightarrow_L)$ of requests.

Here, L is the set of requests received by a server and $\rightarrow_L$ is the order in which those requests were received. Only update requests are actually logged. Read only requests are omitted because they do not affect an object's state. Note that because servers may receive requests in different orders, they may also log requests in different orders.

Definition 2.2 The *projection* of a log, $(L, \rightarrow_L)$, onto an object, $A \in D$, is the set of object A requests in the log. Formally,

$$(L, \rightarrow_L) \mid_A \; = \; \{ \; x \in L \mid x \text{ is a request on object } A \; \}$$

In order to decouple the execution speed of servers from the speed of logs, servers maintain their logs asynchronously. No coordination occurs between the logs of different servers. In addition, no coordination occurs between the state of a server and its log. The state of a log may often lag behind the state of its server. (This approach is orthogonal to that of *write-ahead logging* where the state of an object and its log are always synchronized [BHG87].)

2.3 Failures

Servers fail by *crashing* [SS83]. When a server crashes, it immediately ceases to receive, process, and log client requests. We will not address the problem

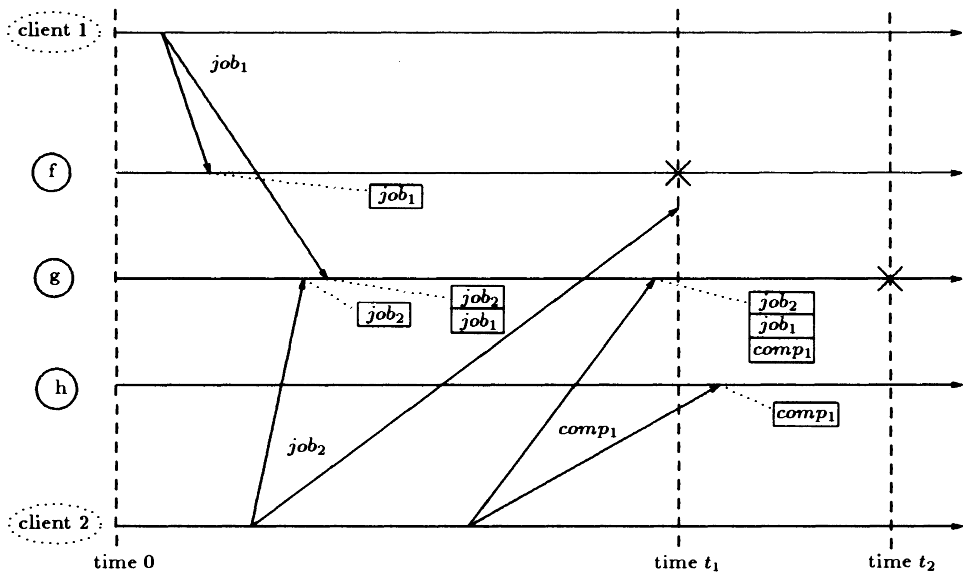

Figure 3: One possible execution of the printer service of figure 2. Depicted are two job submission broadcasts (job_1 and job_2) along with one job completion notification ($comp_1$). Also depicted are two failures. Server f fails at time t_1 and server g fails at time t_2. In the diagram, horizontal lines represent process (server or client) executions while diagonal arrows represent request message broadcasts. Dotted lines represent the logging of request messages by servers. The length of a dotted line indicates the latency between the receipt of a request and its physical logging.

of server partitions [DGMS85]. When a server is functioning, we assume that it can communicate with all other functioning servers.

Figure 3 depicts a possible execution of the printer service shown in figure 2. In the example, server f fails at time t_1 after receiving (and logging) job_1, but before receiving job_2. Server g fails at time t_2 after receiving (and logging) all three requests. Server h functions continuously through out the example, receiving and logging the job completion notification $comp_1$. Note that the final logs of servers f and g do not agree on the state of object $Jobs$. Not only do they contain different requests for the object, but they reflect different orders on those requests.

2.4 System State

At the time a server recovers, the objects in the system can be divided into two categories. An *active* object is one for which some server is actively managing a replica. An *inactive* object is one for which all servers of the object have failed or are in the process of restoring their replicas.

For the purpose of recovery, the state of the system can be summarized in the following manner:

Definition 2.3 A *state* of the system is characterized by the following values:

For each data object, $A \in D$:

- ACT_A The set of servers actively managing a replica of object A.
- REC_A The set of servers in the process of restoring their replicas of object A.
- FAL_A The set of failed servers of object A.

For each server, $f \in SERV$:

- $(L_f, \rightarrow_f)$ The log of server f.

$$
\begin{array}{lll}
ACT_{Jobs} = \{g\} & REC_{Jobs} = \emptyset & FAL_{Jobs} = \{f\} \\
ACT_{Comps} = \{g, h\} & REC_{Comps} = \emptyset & FAL_{Comps} = \emptyset
\end{array}
$$

$$
(L_f, \rightarrow_f)\colon \boxed{job_1} \qquad (L_g, \rightarrow_g)\colon \boxed{\begin{array}{c} job_2 \\ \hline job_1 \end{array}} \qquad (L_h, \rightarrow_h)\colon \boxed{comp_1}
$$

Figure 4: A state from the printer service execution given in figure 3. Depicted is the state of the system immediately after time t_1.

As an example, consider again the execution of figure 3. Figure 4 shows the state of this system immediately after time t_1.

3 Causal Dependencies

During the execution of a system clients can interact with one another.[1] These interactions often lead to data dependencies between the requests the clients issue. For example, in figure 2 the job completion notification $comp_1$ is causally dependent[2] on the job submission job_1: a job cannot complete until *after* it has been submitted.

Causal dependencies restrict the set of correct request orderings that can be perceived by servers. A server should never receive two causally related requests out of causal order. Earlier it was stated that servers may receive requests in differing orders, provided that those orders are correct for the application. This can be stated more precisely by saying that servers may receive requests in any order consistent with causality.

[1] Clients can interact either directly, by sending messages to one another, or indirectly, through the objects managed by the servers.

[2] Many types of dependencies can exist between client requests. In this paper, however, we will focus on *causal dependencies*.

$$\text{Request System: } (R, \prec_R)$$
$$R = \{job_1,\ job_2,\ comp_1\}$$
$$job_1 \prec_R comp_1$$

Figure 5: A request system representing the dependencies in the printer service. The system consists of three requests: two job submissions and a job completion notification. The only causal dependency in the system is the one between the completion notification of job_1 and its submission.

3.1 Request Systems

The causal dependency structure of an application can be summarized by means of a *request system.*

Definition 3.1 A *request system* is a partially ordered set $(R, \prec_R)$ of requests.

Here, R is the set of all requests made by clients in the system and $\prec_R$ is a partial order that relates all pairs of causally dependent requests. The partial order $\prec_R$ may be interpreted as meaning that if two requests are related, $x \prec_R y$, then request y is causally dependent on request x (*i.e.* request y must follow request x). The relation $\prec_R$ is equivalent to the "*happens before*" relation of Lamport [Lam78]. Like the "*happens before*" relation, $\prec_R$ is transitive. We will sometimes use the notation $x.A$ in order to refer to a client request made on object A.

Figure 5 shows the request system for the example given in figure 2. Note that causal dependencies hold between requests made by different clients as well as between requests made on different objects: request $comp_1$ is dependent on request job_1, even though the former is made by *client* 1 on object *Jobs* while the latter is made by a different client on a different object.

It is the responsibility of clients to enforce any request ordering constraints that must hold between their requests. Servers simply process and log requests in the order in which they are received. One possibility is for clients to use reliable ordered broadcasts [BJ87b,CM84,CASD86] to ensure the proper ordering between requests.

3.2 Dependencies and server logs

Because servers log requests in the order they are received, casual dependency constraints also apply to logs. That is, the ordering of requests in a server's log should always be consistent with the request ordering constraints of the application. This observation can be formalized as follows:

Definition 3.2 A log, $(L_f, \rightarrow_f)$, for server f is *causally consistent* with respect to a request system, $(R, \prec_R)$, if

$$\begin{aligned} &\forall y.B \in L_f : \ \forall x.A \in R \ (x \prec_R y) : \\ &\qquad\qquad (f \in SERV_A) \implies (x.A \in L_f \ \wedge \ x \rightarrow_f y) \end{aligned}$$

3.3 Dependencies and recovery

Causal dependencies also affect the issue of consistency between the states of different objects. The state of a system should never reflect a request unless all of the causal dependents of that request are also reflected in the state of the system. For example, the printer service should not reflect the completion ($comp_1$) of the first job unless it reflects the job's submission (job_1). Insuring this type of consistency is the problem at the heart of object replica recovery. The problem is analogous to the problem of generating checkpoints along a consistent cut.

3.4 Maintaining dependency information

Many methods exist for maintaining causal dependency information about the updates in a system. One method is to tag each update with a list of identifiers of its dependents; this is the approach taken in Psync [PBS88].

A similar method, and the one used in ISIS [BJ87b], is to piggyback each update message with a copy of each of its dependents. Another method is to tag each update with a timestamp that reflects the update's causal ordering with respect to other updates; this approach is used in both the highly available services [LL86] and optimistic failure recovery [SY85,JZ88]. Each of these examples illustrates a method based on maintaining dependency information explicitly. Unfortunately, it can be difficult or impossible to maintain explicit dependency information when the set of clients is either unknown or large and dynamically changing. In this paper we examine an alternate approach based on maintaining dependency information implicitly. In particular, dependency information is estimated from the ordering of updates in servers' logs.

4 Failure Recovery

Servers use their logs to recover from failures in the usual way. In order to reconstruct the state of a failed object replica, a recovering server simply re-executes the sequence of requests logged for that object. Once the state of the object replica is restored, the recovering server begins receiving and processing new requests for it. Several synchronization problems potentially arise, however, if the states recovered by servers are not coordinated.

4.1 Synchronization Problems

Because of request dependencies and uncoordinated logs, a failed server can recover its replica of an object in a state that is inconsistent with the states of other object replicas in the system. We present three examples to illustrate how such inconsistencies can occur.

One type of inconsistency can occur when a failed server recovers its replica of an object that is already active in the system. The state of the replica recovered by the failed server will be the state of the object from the time of the server's failure. Since the time of this failure, however, the object has probably undergone changes that will be reflected in the states of the active replicas. The state recovered by the failed server will, therefore, likely disagree with the active replicas. This problem can be illustrated in figure 3. Suppose server f recovers between time t_1 and time t_2. The state it recovers for object $Jobs$ (the state represented in its log) does not reflect the submission of job_2. Server f will therefore disagree with server g on the set of submitted jobs.

This problem can be easily solved by transferring the state of the active object replicas to the failed server at the time of its recovery. The recovering server could then ignore its log and use the transferred state to initialize its object replica. This is the approach used by ISIS [BJ87a] and will be the approach taken here. We refer to the problem of initializing replicas of active objects as the JOIN problem. A more formal discussion of this problem is given below.

A similar type of inconsistency can occur when several failed servers all simultaneously attempt to recover their replicas of the same *inactive* object. Because each server probably failed at a different time, each server's log probably reflects a different state for the object. It is therefore likely that each server will recover its replica in a state that disagrees with the states recovered by the other servers. This problem can also be illustrated in figure 3. Suppose both server f and server g recover after time t_2. In this case, server g will recover submission job_2 while server f will not.

In order to solve this problem, the recovering servers must cooperate and agree on a state for the inactive object. Ideally, this state should be as recent as possible. In synchronous systems, where the states of replicas are coordinated, the most recent logged state is that of the last server to fail [Ske85]. However, in asynchronous systems, this is not true. Any server may have potentially logged the most recent state. It is even possible that different servers may have logged different requests. In this case, none of the logged states is the most recent. Each contains some requests that are not present in the other logs. A fairly recent state can generally be constructed, though, by merging the logs of all recovering servers.

Both of the above examples illustrate synchronization problems that are rooted in the asynchrony of logging and failures. A more difficult synchronization problem arises from the presence of request dependencies. As shown above, an object replica can be recovered in a variety of states, depending on who is recovering the replica and at what time. Because of this, it is possible that replicas of two different objects can be recovered in causally *inconsistent* states. That is, one object can be recovered in a state that contains a request for which causal dependents (on other objects) were not recovered.

For example, consider figure 6. This figure shows an execution of the printer service similar to the one given earlier. However, unlike in the earlier execution, server f fails before logging any request, and server h fails at time t_2 (in addition to server g). If servers f and h were both to recover (after time t_2) before server g, the system would be in a causally inconsistent state. That is, the system state would reflect $comp_1$, the completion of job_1, without reflecting the submission of job_1.

4.2 Synchronization Phase

In order to solve these problems, the log of a recovering server can be synchronized with the logs (states) of the other servers in the system at the time of recovery. The recovering server's log can by synchronized with the logs of active servers (on the states of active objects) as well as with the logs of other recovering servers (on the states of inactive objects).

We divide the recovery sequence of a failed server into two parts: a JOIN part and an ACTIVATE part. The JOIN part addresses the problem of synchronizing the recovering server's log with the states of active objects. The ACTIVATE part addresses the problem of synchronizing the recovering server's log with the logs of other recovering servers on the states of inactive objects. Figure 7 illustrates the relationship of these two parts in the recovery sequence.

The JOIN and ACTIVATE problems are formally described below and

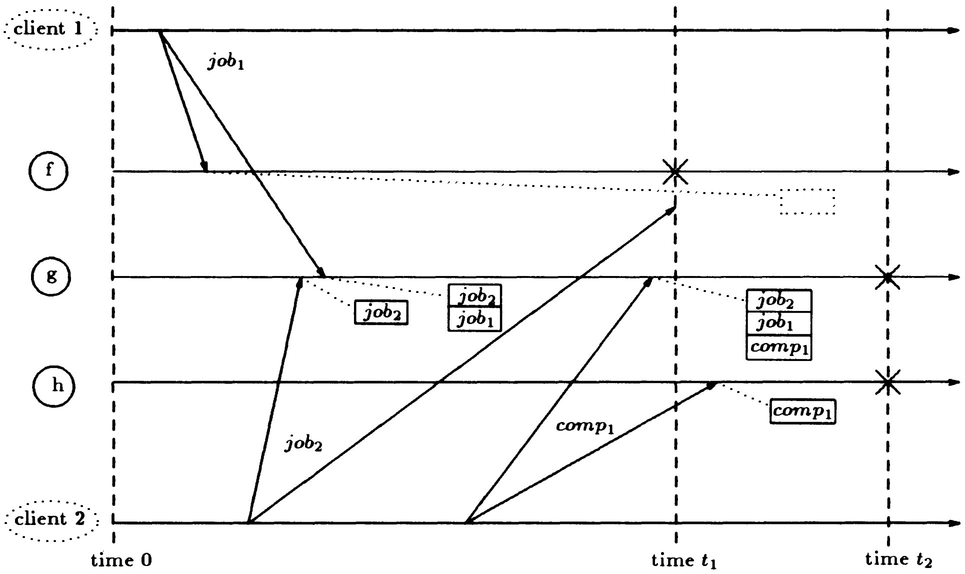

Figure 6: An example of causally inconsistent recovery. If servers f and h were both to recover after time t_2, they would recover in mutually inconsistent states (server h would reflect the completion, $comp_1$, of job_1 while server f would fail to reflect the job's submission). Note the dotted box near the time line of server f. This box shows the point at which the job submission job_1 would have been logged by server f, had that server remained functioning.

their solutions are presented in the following sections. In order to simplify the discussion we will assume that, at the time of a server recovery, all active servers of an object have received and logged the same set of requests for that object (although possibly in different orders). We will refer to this set of requests as the *active state* of the object. Formally, the active state of object A is

$$AS_A = (L_f, \rightarrow_f) \mid_A \quad \forall f \in ACT_A$$

This assumption on the states of active servers may appear to violate the statement that servers receive and log requests asynchronously. However, the assumption only applies to the active servers of an object and only at times of a server recovery. It should be pointed out that enforcing the assumption is relatively easy. The details can be found in [Kan89].

JOIN Problem

When a server, $f \in SERV$, first recovers from a failure, its log is brought into agreement with the states of the active objects[3] in the system. The recovering server's old log, $(L_f, \rightarrow_f)$, is altered to create a new log, $(L_f^*, \rightarrow_f^*)$, that agrees with the logs of active servers on the states of active objects. Formally, a new log for server f is generated with the following properties:

- The new log is causally consistent.
- The new log is in agreement with the states of objects that are active. That is,

$$\forall A \in D \ (ACT_A \neq \emptyset) : \ f \in SERV_A \implies (L_f^*, \rightarrow_f^*) \mid_A = AS_A$$

ACTIVATE Problem

Once the log of a recovering server has been synchronized with the states of the active objects in the system, it is synchronized with the logs of the

[3]An object $A \in D$ is active if $ACT_A \neq \emptyset$.

other recovering servers in the system on the states of inactive objects. This synchronization is done one inactive object at a time.

Let A denote an inactive object that is being recovered. All of the recovering servers of object A (all of the members of REC_A) participate in the recovery of object A. A new state is chosen for object A that is consistent with the states of the active objects in the system. Each of the recovering servers then installs this state in its log as the state of object A. More precisely, the old logs of the recovering servers

$$\{ (L_f, \rightarrow_f) \mid f \in REC_A\}$$

are altered to create new logs

$$\{ (L_f^*, \rightarrow_f^*) \mid f \in REC_A\}$$

that are in agreement on the state of object A. Formally, the new logs generated for the recovering servers will have the following properties:

- Each new log, $(L_f^*, \rightarrow_f^*)$, is causally consistent.

- All new logs agree on the state of object A. That is,

$$\forall\, f, g \in REC_A : \quad (L_f^*, \rightarrow_f^*) \mid_A = (L_g^*, \rightarrow_g^*) \mid_A$$

- The state of object A reflected in the new logs is causally consistent with the states of *all* active objects in the system. That is, for any active object, B,

$$\forall\, x.A \in (L^*, \rightarrow^*) \mid_A : \ \forall\, y.B \in R \ (y.B \prec_R x.A) : \ y.B \in AS_B$$
$$\text{and}$$
$$\forall\, y.B \in AS_B : \ \forall\, x.A \in R \ (x.A \prec_R y.B) : \ x.A \in (L^*, \rightarrow^*) \mid_A$$

  where $(L^*, \rightarrow^*)$ is any of the new logs.

- If a server, $f \in REC_A$, is actively managing a replica of some object, B, then the new log does not interfere with the state logged for that active object. That is,

$$\forall\ f \in REC_A :\ \forall\ B \in D\ (ACT_B \neq \emptyset) :$$
$$(L_f^*, \rightarrow_f^*)\ |_B = (L_f, \rightarrow_f)\ |_B$$

Note that all servers participating in this synchronization phase should have previously completed their JOIN phases. The JOIN phase provides each participating server with information about the states of active objects in the system. This information is used in the ACTIVATE phase in order to ensure that the state recovered for the inactive object is causally consistent with the states of active objects.

Examples

As an example of JOIN and ACTIVATE consider figure 6. Suppose that server h is the first server to recover after time t_2. No objects will be active at the time h recovers. The JOIN phase of server h will not therefore need to take any synchronization actions. Server h will, however, ACTIVATE object *Comps* by replaying its log, restoring its replica of *Comps* to a state reflecting the completion of job_1. Now, suppose server f recovers next. Again, no synchronization actions will be taken in the JOIN phase of f because the object that it servers (*Jobs*) is inactive at the time of f's recovery. Server f will therefore proceed to ACTIVATE object *Jobs* by replaying its log. Note that the state of *Jobs* reflected in f's log is inconsistent with the active state of object *Comps*. In order to restore *Jobs* to a state *consistent* with *Comps*, f will add request job_1 to its log before replaying it. If server g then recovers last, both of the objects it serves will be active. It's JOIN phase will therefore consist of synchronizing its log with both of these objects. Server g accomplishes this by deleting request job_2 from its log.

As another example, suppose that server h is not the first server to recover after time t_2. Instead, suppose that server f is the first to recover. Again, no actions are taken during f's JOIN phase. Server f thus proceeds to ACTIVATE object *Jobs* by replaying its log. Note that, unlike before, object *Comps* is not active at this time. The state of *Jobs* recovered by

Recovery Sequence of Server f:

JOIN Phase:

1. **for each** $A \in D : ACT_A \neq \emptyset \wedge f \in SERV_A$:
   choose an active server, g, of object A
   synchronize $(L_f, \rightarrow_f)$ with $(L_g, \rightarrow_g)$ on the state of A
2. reconstruct replicas of active objects from $(L_f, \rightarrow_f)$
3. begin processing new requests on active objects

ACTIVATE Phase:

4. **while** $\exists A \in D \ : \ ACT_A = \emptyset \wedge f \in SERV_A$:
   form a new state for object A by merging the logs of
   all of its recovering servers (REC_A)
   if the new state is inconsistent with the state of any
   active object B $(ACT_B \neq \emptyset)$ **then** abort the
   activation of A until additional servers recover
   $\forall \ g \in REC_A$: Install the new state in the log,
   $(L_g, \rightarrow_g)$, of server g
   reconstruct replica of object A from $(L_f, \rightarrow_f)$
   begin processing new requests on object A

Figure 7: Recovery Outline

f is not therefore required to be consistent with *Comps*; server f is free to restore *Jobs* to a state that does not reflect the submission of any jobs. Now, suppose that server g is the next to recover. During its JOIN phase, server g will synchronize its log with that of server f on the state of *Jobs*. To do this, g will delete both job submissions from its log. In addition, g will also delete the completion notification ($comp_1$) in order to preserve the causal consistency of its log. Once g has restored its replica of *Jobs* it will proceed to ACTIVATE object *Comps*, restoring its replica of that object to a state that does not reflect the completion of any jobs. When server h finally recovers, it will delete $comp_1$ from its log and restore its object replica to the appropriate state.

5 Log Transformations

Our algorithms implementing the JOIN and ACTIVATE phases are based on functions for *adding* and *deleting* requests from servers logs. The main difficulty in designing these functions is ensuring that they preserve the causal consistency of the logs on which they operate.

5.1 Log Addition

Consider first the problem of adding a request to a log. Let $x.A$ denote a request on object A and let f denote a server of object A (*i.e.* $f \in SERV_A$).

Request $x.A$ is added to the log of server f, $(L_f, \rightarrow_f)$, by inserting it into the log at some point where the resulting log order remains consistent with $\prec_R$. The resulting log, however, is not necessarily causally consistent. There may be causal dependents of request $x.A$, on objects served by f, that are not present in the resulting log. In order to preserve the causal consistency of server f's log, these *missing dependents* must also be added.

Let $DEP_B(x.A)$ denote the set of requests on object B that are causal dependents of request $x.A$. Formally,

$$DEP_B(x.A) \;=\; \{y.B \in R \mid y \prec_R x\}$$

We denote the function of adding request $x.A$ to the log of server f as $\mathbf{add}_{x.A}(L_f, \rightarrow_f)$. Formally, this function is defined as:

$$\mathbf{add}_{x.A}(L_f, \rightarrow_f) \;=\; (L, \rightarrow_L)$$

where

$$L \;=\; L_f \cup \{x.A\} \cup [\bigcup_{\{B \mid f \in SERV_B\}} DEP_B(x.A)]$$

$\rightarrow_L$ is any extension of $\rightarrow_f$ consistent with $\prec_R$.

This definition can easily be extended to accommodate the addition of multiple requests. We will let $\mathbf{add}_Q(L_f, \rightarrow_f)$ denote the addition of all of the requests in Q to the log of server f. The definition of $\mathbf{add}_Q(L_f, \rightarrow_f)$ can be found in [Kan89].

5.2 Log Deletion

The deletion of a request from a server's log is handled in a manner analogous to the addition of a request. As above, let $x.A$ denote a request on object A and let f denote a server of object A.

Request $x.A$ is deleted from the log of server f by simply removing it, preserving the order of the remaining requests. Again, however, the resulting log may not be causally consistent. There may be requests remaining in the log that are dependent on the deleted request. In order to preserve the causal consistency of the log, these requests must also be removed.

Let $CON(x.A \prec y.B)$ denote the relation that request $y.B$ is *not* causally dependent on request x.A. That is, the relation $CON(x.A \prec y.B)$ is true when $x.A \not\prec_R y.B$ holds (the relation is contradicted).

We denote the function of deleting request $x.A$ from the log of server f as $\mathbf{delete}_{x.A}(L_f, \rightarrow_f)$. Formally, this function is defined as:

$$\mathbf{delete}_{x.A}(L_f, \rightarrow_f) \;=\; (L, \rightarrow_L)$$

where

$$L \;=\; \{y.B \in L_f \mid y.B \neq x.A \;\wedge\; CON(x.A \prec y.B)\}$$
$$\forall\, x, y \in L : \; (x \rightarrow_L y) \Leftrightarrow (x \rightarrow_f y)$$

As was the case for log addition, log deletion can easily be extended to accommodate the deletion of multiple requests. We let $\mathbf{delete}_Q(L_f, \rightarrow_f)$ denote this function. Its definition can also be found in [Kan89].

6 Synchronization Solutions

JOIN and ACTIVATE are implemented by adding and deleting requests from a server's log. A recovering server's log is altered until it reflects a state that is in agreement with the states of the other servers in the system. The log is then used by the recovering server to reconstruct the states of its object replicas.

6.1 JOIN Implementation

When a server, f, first recovers from a failure its log, $(L_f, \rightarrow_f)$, is brought into agreement with the states of active objects. The current states of the active objects are transferred to the recovering server and written in its log, replacing any states previously logged for the objects. The actual log of server f, $(L_f, \rightarrow_f)$, is altered in two ways. First, any request for an active object that is present in the log, but not present in the object's transferred state, is removed from the log. These requests represent updates that were never recovered for the object. Formally, these *non-recovered* requests are:

$$NR_f \;=\; \bigcup_{\{A \mid ACT_A \neq \emptyset\}} [\, (L_f, \rightarrow_f) \mid_A \;-\; AS_A \,]$$

Second, any request present in the transferred state of an object, that is not present in the log, is added to the log. These requests represent updates that were missed by the recovering server during its failure. Formally, these *missing* requests are:

$$MS_f \;=\; \bigcup_{\{A \mid f \in SERV_A\}} [\, AS_A \;-\; (L_f, \rightarrow_f) \mid_A \,]$$

The resulting log, the log solving the JOIN problem for server f, is:

$$(L_f^*, \rightarrow_f^*) \;=\; \mathbf{add}_{MS_f}(\mathbf{delete}_{NR_f}(L_f, \rightarrow_f))$$

6.2 ACTIVATE Implementation

Once the log of a recovering server is brought into agreement with the states of active objects, it is then brought into agreement with the logs of other recovering servers on the states of inactive objects. Let A denote an inactive object in the system (*i.e.* $ACT_A = \emptyset$). And let A be such that all recovering servers (*i.e.* all members of REC_A) have completed their JOIN phases.

In order to restore object A, the recovering servers of A first agree on a state for it. Ideally, this state is the most complete state constructible from their logs, the state formed by combining all of their logged requests:

$$IS_A \;=\; \bigcup_{f \in REC_A} (L_f, \rightarrow_f) \mid_A$$

This *ideal state* can, however, be inconsistent with the states of some active objects. There may be requests in the ideal state that have dependencies on requests for other objects that were not recovered for those objects. In order to preserve the consistency between objects, these inconsistent requests must be omitted from the recovered state of object A.

We let $SAFE(x.A)$ denote the predicate that request $x.A$ is consistent with the states of all active objects. That is, request $x.A$ does not have any dependencies on requests not recovered for those objects. Formally,

$$SAFE(x.A) \;\equiv\; \bigwedge_{\{B \;\mid\; ACT_B \neq \emptyset\}} [\; DEP_B(x.A) \subseteq AS_B \;]$$

The state recovered for object A, the most complete *and consistent* state constructible from the server's logs, is therefore:

$$NS_A \;=\; \{x.A \in IS_A \mid SAFE(x.A)\}$$

This state is installed into the logs of the recovering object A servers in the same manner that the transferred states were installed into their logs during their JOIN phases. For each recovering server, f, the new state is installed in two parts. First, any object A request present in the log of server f, that is not present in the new state, is removed from the log. These removed requests are the inconsistent requests that were omitted from the ideal state. Formally, these requests are:

$$NR_f \;=\; (L_f, \rightarrow_f)\mid_A \;-\; NS_A$$

Second, any request present in the new state, that is not present in the log, is added to the log. Formally, these *missing* requests are:

$$MS_f \;=\; NS_A \;-\; (L_f, \rightarrow_f)\mid_A$$

The resulting log, the log solving the ACTIVATE problem for object A at server f, is:

$$(L_f^*, \rightarrow_f^*) \;=\; \mathbf{add}_{MS_f}(\mathbf{delete}_{NR_f}(L_f, \rightarrow_f))$$

Actually, the new state recovered for object A may not be totally consistent with the states of active objects. It is possible that an active object may have a dependency on a request that is not recovered for object A. This can happen, for example, if the dependent request was never logged or because the servers that did log it never recovered in time to take part in the ACTIVATE phase. When this problem of a *missing dependent* occurs, the ACTIVATE phase must abort the restoration of object A. It must then wait for additional servers of object A to recover (hopefully with the missing dependent present in one of their logs) before re-attempting to activate the object.

7 Dependency Estimation

The implementations of the JOIN and ACTIVATE phases assume that servers have knowledge of $\prec_R$. In particular, the implementations are based on the values of $DEP_B(x.A)$, $CON(x.A \prec y.B)$, and $SAFE(x.A)$, which depend on $\prec_R$. Servers, however, will not often have access to this dependency information. Thus, servers will not be able to use the implementations as they have been presented. Instead, servers will have to estimate dependency information and use those estimates to coordinate their logs.

7.1 Dependency Types

Request dependencies can be estimated from the orderings of requests in servers logs. There are two types of dependencies: *transitive* and *direct*. A transitive dependency is a dependency formed from the composite of other dependencies. That is, a dependency, $x \prec_R y$, is transitive if it is due to a sequence of direct dependencies:

$$x \prec_R z_1 \prec_R z_2 \prec_R \dots \prec_R z_n \prec_R y$$

A dependency is direct if it is not the composite of other dependencies. Direct dependencies are the basic dependencies in a system. Formally,

Definition 7.1 An dependency, $x.A \prec_R y.B$, is *direct* if

$$\neg \exists z \in R : \ (x \prec_R z \bigwedge z \prec_R y)$$

Definition 7.2 A dependency, $x \prec_R y$, is *transitive* if it is not direct.

7.2 Object Dependency Relation

As stated above, servers will not often have knowledge of the causal dependency relation. We assume, though, that servers have knowledge of a generalization of this relation. This generalization is called the object dependency relation.

Definition 7.3 The *object dependency relation*, $A \rightsquigarrow_R B$, holds between two objects, $A, B \in D$, if it is possible that an object B request is causally dependent on an object A request.

The object dependency relation only tells a server about *potential* dependency. If the relation $A \rightsquigarrow_R B$ holds, then it is possible that some object B requests are dependent on some object A requests. However, a server does not know which requests, if any, are related. If two objects are unrelated, $A \not\rightsquigarrow_R B$, then a server does know that no object B request is causally dependent on any object A request. Formally, this can be summarized as follows:

$$\forall\ x.A, y.B \in R: \quad x.A \prec_R y.B \implies A \rightsquigarrow_R B$$

7.3 Basic Estimates

We begin this section by presenting our basic estimates of the causal dependency relation. These estimates are designed to approximate *direct* relationships between requests. The basic estimates are used later in this section to build more complex estimates for approximating *transitive* relationships.

In order to help ensure that each direct dependency is represented in the order of some server's log, we assume that any pair of objects, between which direct dependencies may hold, have overlapping server sets. Formally, we assume that

$$\forall \text{ direct } x.A \prec_R y.B : \ (SERV_A \bigcap SERV_B \neq \emptyset)$$

7.3.1 Request Ordering

Our basic estimate of the relation $CON(x.A \prec y.B)$ is denoted by the relation $con^0(x.A \prec y.B)$. It is designed to approximate whether or not two requests, $x.A$ and $y.B$, are directly related. In particular, when the estimate $con^0(x.A \prec y.B)$ is true, it is guaranteed that request $y.B$ is *not* causally dependent on request $x.A$. When the estimate is false, though, there is no guarantee as to whether or not the requests are related.

The idea behind the estimate is to examine servers' logs for evidence that the requests are unrelated. Recall that, according to the causality constraint on logs, if a server logs one request then it must previously have logged all of that request's dependents that are on objects managed by the server. It therefore follows that if some server has logged request $y.B$ before request $x.A$, then $y.B$ cannot be causally dependent on request $x.A$. Similarly, if some server of both objects A and B has logged request $y.B$ but not request $x.A$, then request $y.B$ cannot be dependent on request $x.A$. Formally,

Definition 7.4 The relation $con^0(x.A \prec y.B)$ holds between any two requests, $x.A, y.B \in R$, if and only if any of the following conditions is true:

1. $A \not\mapsto_R B$
2. $\exists\ f \in SERV_A \bigcap SERV_B :\ \ x.A, y.B \in L_f\ \bigwedge\ y.B \rightarrow_f x.A$
3. $\exists\ f \in SERV_A \bigcap SERV_B :\ \ y.B \in L_f\ \bigwedge\ x.A \notin L_f$

7.3.2 Dependency Set

Our basic estimate of $DEP_B(x.A)$ is denoted $dep^0_B(x.A)$. It is designed to approximate the set of direct object B dependents of request $x.A$. Our estimate has the property that, when defined, it is either equal to the true dependency set or an overestimate of it.

Like the previous estimate, this estimate is based on the causality constraint for logs. As pointed out earlier, if a server logs one request, then it must previously have logged all of that request's dependents that are on objects it serves. Thus, if a server of both objects A and B has logged request $x.A$, then that server's log must contain all of the object B dependents of $x.A$ in positions preceding $x.A$ in the log. The set of object B requests preceding $x.A$ can therefore be used as an estimate of the true dependents.

Some of these object B requests may not, however, be real dependents of $x.A$. They may just be requests that happened to get logged before $x.A$. Some of these extraneous requests can be detected and eliminated from the approximation by using the first basic estimate. In particular, the set of object B dependents of request $x.A$ can be estimated as follows:

$$dep^0_B(x.A) = \begin{cases} \perp & \text{if } \neg\exists f \in SERV_A \bigcap SERV_B : \; x.A \in L_f \\ \emptyset & \text{if } B \not\rightsquigarrow_R A \\ \{y.B \mid \exists f \in SERV_A \bigcap SERV_B : & \\ \quad (x.A, y.B \in L_f \;\bigwedge\; y.B \rightarrow_f x.A & \text{o.w.} \\ \quad \bigwedge\; \neg con^0(y.B \prec x.A)) \;\} & \end{cases}$$

Note that the estimate is undefined when no server of both objects A and B has logged request $x.A$. Under this condition, the estimation method presented here cannot be used.

7.4 Compound Estimates

Transitive dependencies are estimated by approximating the sequences of direct dependencies out of which they are built. In presenting these estimates, the following definition will be useful:

Definition 7.5 A *chain*, H, is a sequence of related objects.

$$H \;=\; A_1 \rightsquigarrow_R A_2 \rightsquigarrow_R \ldots \rightsquigarrow_R A_n$$

Intuitively, a chain represents a sequence of objects along which a transitive dependency may occur. If a chain such as H exists, then it is possible for an object A_n request to be dependent on an object A_1 request through a sequence of dependencies on requests on objects A_{n-1}, A_{n-2}, $\ldots$, A_2. However, there is no guarantee that such a transitive dependency exists. The existence of a chain only implies the potential for such a dependency. We let $AB\text{-}CHAINS$ denote the set of all chains from object A to object B.

The following definitions, based on chains, will also be useful:

Definition 7.6 A *sub-chain* of a chain, H,

$$H = A_1 \leadsto_R A_2 \leadsto_R \ldots \leadsto_R A_n$$

is any subsequence of its objects

$$H' = A_{m_1} \leadsto_R A_{m_2} \leadsto_R \ldots \leadsto_R A_{m_p}$$

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

Definition 7.7 The A_iA_j *sub-chain* of a chain, H,

$$H = A_1 \leadsto_R A_2 \leadsto_R \ldots \leadsto_R A_n$$

is the sub-chain of objects from A_i to A_j:

$$H_{i..j} = A_i \leadsto_R A_{i+1} \leadsto_R \ldots \leadsto_R A_j$$

7.4.1 Dependency Set

We denote our compound estimate of $DEP_B(x.A)$, the object B dependents of request $x.A$, by the request set $dep^{\omega}_B(x.A)$. This estimate, like the basic estimate, has the property that when defined it contains all of the object B dependents of request $x.A$, plus possibly a few extraneous requests.

This estimate is built out of estimates of dependencies along individual chains. In order to estimate the object B dependents of request $x.A$, the dependents along each chain from A to B are separately estimated. These estimates are then combined to form a complete estimate of the dependency set. Specifically, let H denote any chain.

$$H = A_1 \leadsto_R A_2 \leadsto_R \ldots \leadsto_R A_n$$

We let $dep^{\omega}_H(x_n.A_n)$ denote our estimate of the object A_1 dependents of request $x_n.A_n$ that occur along chain H. That is, $dep^{\omega}_H(x_n.A_n)$ estimates the set of object A_1 requests that are related to $x_n.A_n$ by a sequence of dependencies on the objects in chain H.

The estimate $dep^{\omega}_{H}(x_n.A_n)$ can be formed in many ways. First, if there is a server that manages replicas of both objects A_1 and A_n, then an estimate can be obtained by simply applying the basic estimate $dep^{0}_{A_1}(x_n.A_n)$. 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 subdividing the problem. That is, an estimate can be formed by first choosing some object in the chain, A_i $(1 < i < n)$, estimating the object A_i dependents of $x_n.A_n$, and then estimating the object A_1 dependents of the object A_i dependents. Again, if the server sets of objects A_1 and A_i overlap, and the server sets of objects A_i and A_n overlap, the basic estimates can be applied to solve each these sub-problems. That is, if the server sets overlap, an estimate can be formed directly along the sub-chain

$$A_1 \leadsto_R A_i \leadsto_R A_n$$

However, this is not likely to be the case unless the pairs of objects are directly related. If the server sets do not overlap, then each of the sub-problems will have to be further subdivided in a manner similar to the original problem. In general, the problem will have to be sub-divided until a sub-chain of H is found

$$A_1 \leadsto_R A_{m_1} \leadsto_R A_{m_2} \leadsto_R \ldots \leadsto_R A_{m_p} \leadsto_R A_n$$

$$1 < m_1 < m_2 < \ldots < m_p < n$$

in which each pair of adjacent objects have overlapping server sets. An estimate can then be formed along this sub-chain by first approximating the object A_{m_p} dependents of $x_n.A_n$; then approximating the object $A_{m_{p-1}}$ dependents of the object A_{m_p} dependents; and similarly approximating the dependents for each successive object down the sub-chain.

This process can be specified recursively in the following manner. Note that the estimate is extended to operate on sets of requests. That is,

$dep^{\omega}_{H}(Q)$ denotes the set of object A_1 dependents of the object A_n requests in Q:[4]

$$dep^{\omega}_{H}(Q) = \begin{cases} \bigcup\limits_{x_2.A_2 \in Q} dep^{0}_{A_1}(x_2.A_2) & \|H\| = 2 \\ \bigcup\limits_{x_n.A_n \in Q} dep^{\omega}_{H_{1..i}}(dep^{\omega}_{H_{i..n}}(x_n.A_n)) & \|H\| > 2 \\ \quad \text{where } 1 < i < n \text{ is chosen so that the estimates are} & \\ \quad \text{defined.} & \end{cases}$$

Note that there may be several choices of i for which the estimates are defined. Each will likely yield a slightly different approximation of the true dependency set. However, each is guaranteed to contain all of the true dependencies that occur along H. Because of this, an even more accurate estimate of the true dependency set (one with fewer extraneous requests) can be formed by intersecting the estimates from each choice of i. The estimate can thus be modified as follows:

$$dep^{\omega}_{H}(Q) = \begin{cases} \bigcup\limits_{x_2.A_2 \in Q} dep^{0}_{A_1}(x_2.A_2) & \|H\| = 2 \\ \bigcup\limits_{x_n.A_n \in Q} [dep^{0}_{A_1}(x_n.A_n) \cap & \\ \qquad [\bigcap\limits_{1<i<n} dep^{\omega}_{H_{1..i}}(dep^{\omega}_{H_{i..n}}(x_n.A_n))\ ]\ ] & \|H\| > 2 \end{cases}$$

Note that the definitions of union and intersection must be extended to take into account the possibility of undefined approximations. This is done as follows:

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

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

[4]The length of a chain, H, is denoted by $\|H\|$. This is the number of objects in the chain.

The estimate of the complete set of object B dependents of request $x.A$ is formed by combining the estimates of dependency along each chain from B to A. Formally,

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

7.4.2 Request Ordering

We denote our compound estimate of the predicate $CON(x.A \prec y.B)$ by the predicate $con^{\omega}(x.A \prec y.B)$. Like the basic estimate, this predicate has the property that, when true, it is guaranteed that request $y.B$ is not casually dependent on request $x.A$. And, when false, the requests may or may not be related.

As with the dependency set estimate, the request ordering estimate is built up from estimates of ordering along individual chains. For any $A_1 A_n$-chain, H, we let $con^{\omega}_{H}(x_1.A_1 \prec x_n.A_n)$ denote our estimate of whether or not request $x_n.A_n$ is causally dependent on request $x_1.A_1$ along chain H. When the estimate is true, it is guaranteed that request $x_n.A_n$ is not dependent on request $x_1.A_1$ by a sequence of dependencies along the chain.

The idea behind this estimate is to search the chain for an object, A_i, at which any possible dependency path from x_1 to x_n is broken. That is, the chain is searched for an object, A_i, such that none of the A_i dependents of x_n are dependent on x_1. The existence of such an object would imply that request x_n is not dependent on request x_1 by a sequence of dependencies that include object A_i. Because A_i is a part of chain H, this would in turn imply that the requests cannot be dependent 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. These dependent requests are then recursively tested to determine if any of them are dependent on $x_1.A_1$. The formal definition of this function is given below. Note that the estimate is extended to operate on sets of requests; that is, $con^{\omega}_{H}(x_1.A_1 \prec Q)$ denotes our approximation of whether or not any of the object A_n requests in Q are causally dependent, along chain H, on request $x_1.A_1$:

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

In order to estimate whether or not two requests are related in general, the estimates of their dependency along individual chains are combined. Formally,

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

7.4.3 Safety

Our last compound estimate is denoted $safe^{\omega}(x.A)$. It is designed to approximate the predicate $SAFE(x.A)$. This estimate has the property that, when true, it is guaranteed that request $x.A$ is consistent with the states of all active objects in the system. That is, when the estimate is true, it is guaranteed that request $x.A$ does not have a dependency on a request for an active object that was not recovered as part of that object's state. If the estimate is false, though, the request may or may not be safe.

Like the other compound estimates, the safety predicate is built from estimates of safety along individual chains. For any A_1A_n-chain, H, we let $safe^{\omega}_{H}(x_n.A_n)$ denote our estimate of the safety of request $x_n.A_n$ along chain H. When true, this estimate guarantees that request $x_n.A_n$ has no dependencies, along chain H, on non-recovered requests for object A_1.

Before defining this estimate, though, some motivation is first presented. Suppose a request $x_n.A_n$ is not safe along some chain, H. That is, request $x_n.A_n$ is dependent on some non-recovered request, $x_1.A_1 \notin AS_{A_1}$, by a sequence of dependencies along chain H:

$$x_1.A_1 \;\prec_R\; x_2.A_2 \;\prec_R\; \ldots \;\prec_R\; x_n.A_n$$

Because each of the requests in this sequence is dependent on $x_1.A_1$, it follows that each request $x_i.A_i$ is also unsafe, along sub-chain $H_{1..i}$. Because unsafe requests are never recovered for an object, it further follows that none of the unsafe requests in the above dependency sequence can be part of their object's active states. Thus, if a request $x_n.A_n$ is unsafe along chain H, then that request has a non-recovered dependent from each active object in the chain. Conversely, if there is an active object in the chain, A_i, whose active state contains all of the object A_i dependents of $x_n.A_n$, then $x_n.A_n$ must be safe along chain H.

The safety estimate is based on looking for objects such as A_i. In particular, the safety of request $x_n.A_n$ along chain H is estimated by examining each active object, A_i, in the chain. For each such object, the dependents of request $x_n.A_n$ are estimated and tested to determine if they are present in the object's active state. If all such dependents are present, then request $x_n.A_n$ is safe along chain H.

$$safe^{\omega}_{H}(x_n.A_n) \;\equiv\; \exists i : \; (ACT_{A_i} \neq \emptyset \;\wedge\; dep^{\omega}_{H_{i..n}}(x_n.A_n) \subseteq AS_{A_i})$$

The general estimate of the safety of request $x.A$ is built by combining the estimates of the request's safety along individual chains. Specifically, a request is safe if it is safe along all chains of dependency from active objects. Formally,

$$safe^{\omega}(x.A) \;\equiv\; \bigwedge_{\{B \in D \;|\; ACT_B \neq \emptyset\}} \; \bigwedge_{H \in BA\text{-}CHAINS} safe^{\omega}_{H}(x.A)$$

7.5 Using the Estimates

The compound estimates can be used directly in the log synchronization algorithms in place of the values they approximate. The proof that the algorithms remain correct is given in [Kan89].

A problem can arise, though, when the estimates do not have access to all of the logs in the system. At the time an estimate is formed, some servers may not be functioning. Because of this, the estimates may have access to limited ordering information, based on which server logs are available. This can lead to undefined estimates, producing aborts of the synchronization algorithms. Unfortunately, there is no way around this problem. When an abort occurs, the server or servers involved must simply wait until additional failed servers have recovered (providing additional ordering information) and then re-execute their synchronization algorithms.

Limited information does not always lead to undefined estimates, however. In order to approximate the dependencies along a particular AB-chain, H, an estimate considers all AB-subchains of H. As long as an approximation can be formed along some sub-chain of H, the estimate will be defined.

Another problem can arise when a synchronization algorithm adds or deletes a request from a server's log. Because the algorithms use only estimates of the dependencies in the system, it is possible that a synchronization algorithm may believe that a request needs to be added or deleted from the state of an active object. When this occurs, it is a sign that the estimates do not have access to sufficient ordering information to deduce accurate approximations. In these cases, the invoking server should abort the synchronization algorithm and wait for additional servers to recover (with additional ordering information for the estimates).

8 Special Systems

When long chains exist in a system, the compound estimates of the previous section can be fairly expensive to compute. In order to form an estimate along a particular chain, H, the compound estimates recursively sub-divide the chain and combine simple estimates from each of the sub-divisions. Unfortunately, the number of sub-divisions of a chain grows exponentially with the length of the chain. Thus, when a chain is long, the number of subdivisions that are considered by the estimates is large. The estimates may therefore be quite expensive to compute. And in turn, the synchronization algorithms will be expensive to run.

In order to reduce this cost, the basic estimates can sometimes be used in place of the compound estimates in the synchronization algorithms. Unlike the compound estimates, the basic estimates do not involve recursion and are, in general, fairly cheap to compute. Unfortunately, the basic estimates can only be used to approximate relationships between two objects, A and B, if the server sets of those objects overlap. Thus, in order to replace the compound estimates with the basic estimates, it must be the case that every pair of related objects have overlapping server sets, regardless of whether the objects are directly or transitively related.

Of course, in general the server sets of all related objects will not overlap, and so the basic estimates will not be able to be used. Even if the server sets do overlap, however, the basic estimates are not guaranteed to be defined at all times. As pointed out in the previous section, server failures can cause estimates to be undefined. The exact estimates that are defined at any given time depends on the servers that are functioning; that is, the estimates that are defined depends on which server logs are available to the estimates.

There is, however, an interesting class of systems in which the basic estimates are always defined. We call this class the backward inclusion systems:

Definition 8.1 A system is a *backward inclusion system* if the following restriction holds on the server sets of objects:

$$\forall\ A, B \in R: \quad A \rightsquigarrow_R B \implies SERV_B \subseteq SERV_A$$

Intuitively, a system is a backward inclusion system if any server that manages a replica of an object, B, also manages replicas of all objects on which B depends. This restriction implies that if a server logs a request $x.B$, then it also logs all dependents of $x.B$.

The set of backward inclusion systems includes hierarchically organized systems such as the one depicted in figure 8. In this figure, each object's server set is completely contained in the server sets of all objects above it in

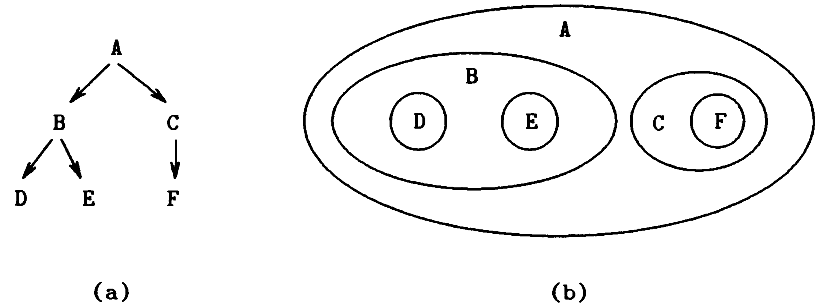

Figure 8: A hierarchical system

the hierarchy. Figure 8(a) shows the tree structured dependency relationship between the six objects in the system. Figure 8(b) shows the overlap between the server sets of the different objects.

The proof that the synchronization algorithms never abort in backward inclusion systems is given in [Kan89]. The proof is based on the fact that each server logs complete dependency information on all of the requests in its log, and so there is always complete dependency information available on any request that is added or deleted from a server's log. It should be pointed out that the log addition and deletion functions must be slightly modified when the basic estimates are used. The reason for this and the appropriate modifications are given in [Kan89].

9 Conclusions

This paper presented a new mechanism for performing optimistic log-based recovery in distributed systems. Unlike existing methods, the mechanism presented does not require the maintenance of explicit dependency information. Instead, by requiring that the server sets of related objects overlap, the mechanism is able to estimate any needed dependency information from the ordering of requests in servers' logs.

In addition, the mechanism avoids the use of process rollback as a synchronization technique. When a server first recovers from failure, its state (the state represented in its log) is brought into agreement with the state of the system. A server is never allowed to recover in an inconsistent state. However, in order to ensure this, a recovering server may have to be blocked until sufficient information is available in the system to deduce that the server's state is consistent. Because of this potential for blocking, a restricted set of systems (the backward inclusion systems) were presented in which blocking never occurs and in which inexpensive dependency estimates can be used.

It should be pointed out that our mechanism makes no guarantees about the consistency between the states of clients and servers when the client and server sets differ. Because of failures, a server may lose a client request. When this happens, our mechanism only ensures that the states of different servers will be brought into agreement. It makes no attempt to coordinate the states of both clients and servers. In some applications, for example the sample printer service described in this paper, the loss of client requests is not critical. In many other applications, however, consistency between clients and servers is crucial. In applications such as these, our mechanism requires that the sets of clients and servers be identical.

Our mechanism can be extended to enforce forms of consistency other than causal consistency. As described in [Kan89], the basic approach of estimating and ensuring dependencies can be used to ensure an *atomic* form of consistency. In the atomic form, a set of requests can be grouped to form a set with the property that no request in the group is recovered after a failure unless all of the requests in the group are also recovered. By combining this atomic form of consistency with the casual form, it may even be possible to derive a *serializable* form of consistency implementable by our basic mechanism.

One problem that remains to be addressed is that of restricting the sizes of logs. As we have presented them, logs can grow without bound. Clearly, in any implementation of the mechanism, the growth of logs must be limited through the use of checkpoints. The main difficulty involved in maintaining checkpoints is estimating the dependencies that exist between different checkpoints and between requests and checkpoints. A detailed discussion of this problem and its solution is provided in [Kan89].

Although the synchronization algorithms presented in this paper have not yet been implemented, we believe that doing so should be fairly straight forward. For example, in the case where the basic estimates are used, object synchronization amounts to little more than sorting. One of the problems with building an implementation, though, is finding applications on which to test and measure its performance. Currently, applications with a high degree of object inter-dependence are rare. Because of the increasing use of object oriented interfaces, however, we believe that such applications will become increasing common.

References

[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.

[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.

[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.

[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.

[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.

[Kan89] Kenneth P. Kane. *Log-Based Recovery in Asynchronous Distributed Systems*. PhD thesis, Cornell University, May 1989. Forthcoming.

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

[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.

[PBS88] Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. Technical Report TR 88-23, Department of Computer Science, University of Arizona, Tucson, AZ 85721, May 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.