A Non-blocking Recovery Algorithm for Causal Message Logging

J. Roger Mitchell* Vijay K. Garg†
email: garg@ece.utexas.edu
Parallel and Distributed Systems Laboratory
Department of Electrical and Computer Engineering
The University of Texas at Austin

Abstract

In the recovery of failed processes in a distributed program, causal logging schemes offer several benefits. These benefits include no rollback of unfailed processes and simple approaches to output commit. Unfortunately, previous approaches to the recovery of multiple simultaneous failures require that the distributed execution be blocked or that recovering processes coordinate. The latter requires assumptions which are not satisfatory. In this paper we present a solution that has neither of these drawbacks.

Message logging is an important technique for recovering from failures in distributed programs. This technique logs the order in which messages are received. By assuming that receive ordering is the only source of non-determinism, execution is recoverable using this ordering. Pessimistic message logging [4, 11] forces a process to wait before sending any message while the message log is written to stable storage. Optimistic logging methods [9, 12, 13, 15] (and the similar sender based logging [8, 14]) assume failures are rare and therefore allow ordering information to be lost in a failure. (That is, a message is logged in the background while execution proceeds). Consequently, received messages and any sends that depend on them may not be recoverable. This may then require that unfailed processes roll back their execution as well. Causal message logging sends message receive ordering information with each message. This information includes receives and their causal history since the last send. The Manetho approach [6] uses this method. In family-based message logging (FBL) [2] causal history information for only K processes is included. This method then tolerates K simultaneous failures rather than all processes in the system (as with Manetho and the other logging methods.)

The causal message logging approach offers advantages over the other message logging schemes. It allows processes to execute without blocking (like optimistic logging) and never forces unfailed processes to roll back their execution (like pessimistic logging).

Unfortunately, causal message logging suffers from complications associated with recovery not present in the other logging methods. One particular difficulty occurs when multiple processes fail simultaneously [7]. Solutions have been presented which require blocking unfailed processes or coordinating between recovering processes. Neither of these solutions is satisfactory. In this paper we present a solution without either of these drawbacks. We note that independently Alvisi, Rao, and Vin have also developed an algorithm for non-blocking recovery [3].

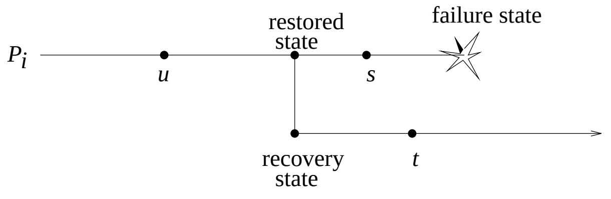
1. Model

Let P_i represent process i and s and t be execution states on a process. Let G represent a global state such that G consists of one state from every process in the computation. Let $G[i]$ represent the state of P_i on G. When we speak of process recovery we mean that a process has been restored to a state in its execution that occurred before a failure. Execution at the process then proceeds from this state. We assume that a process is never permanently disabled such that it never again responds to messages. We also assume that processes do not fail an infinite number of times. Let a *failure state* be the crash of a process. Let a *restored state* be the state from before a failure which is restored upon recovery. Let a *recovery state* be the state following recovery in which the restored state is restarted. Figure 1 illustrates these terms. Let $s \prec t$ if

1. s occurred before t on some P_i and
2. there does not exist a failure k such that s is after the kth restored state and t is after the kth recovery state.

In the figure, $s \not\prec t$, but $u \prec s$ and $u \prec t$.

*supported in part by a Virginia & Ernest Cockrell fellowship
†supported in part by the NSF Grants ECS-9414780, CCR-9520540, Texas Education Board Grant ARP-320, a General Motors Fellowship, and an IBM grant



st

Let $s \preceq t$ mean $s \prec t$ or s equals t. Let the ith *incarnation* be the set of states, S, such that $\forall s \in S$: ith recovery state $\preceq s \prec (i+1)$th recovery state. Let all states s such that, $\forall i$: ith restored state $\prec s \prec$ ith failure state, be rolled back states. In figure 1, s is a rolled back state. Let a *run* consist of all states which are not rolled back. Finally, let $s \rightarrow t$ (the happens-before relation for states) be the smallest relation satisfying:

1. $s \prec t$,
2. s is the send state of a message and t is its receive state,
3. $s \rightarrow u \wedge u \rightarrow t$.

Messages may fail and channels are unordered. When a process failure occurs, it may be restored by restarting at a previous state. If a state s is not restored to the run, a state t is restored to the run, and $s \rightarrow t$, then t is an *orphan state*. To restore the distributed computation without orphan states, if a receive state is in the run, then the corresponding send state must also be in the run. Let S be the set of states, then this property can be given as:

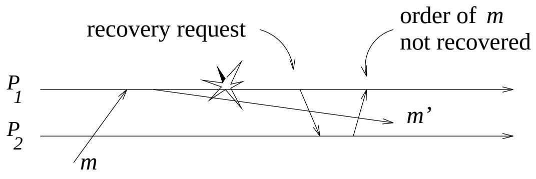
$$\forall s,t \in S : t \in \textit{run} \wedge s \rightarrow t \Rightarrow s \in \textit{run}$$

2. Recovering after failures

When a process fails in causal message logging algorithms, to recover it must recreate its message log (that is, the message receive ordering). The information necessary to encapsulate the ordering of messages is called *determinants* [1, 2]. We will use this concept here.

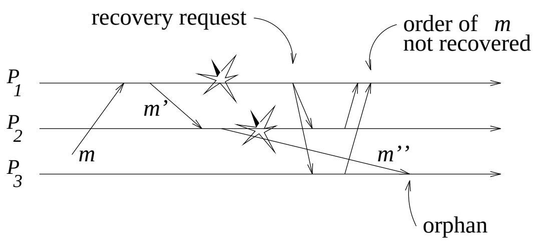
A recovery algorithm for causal message logging must ensure that messages from recovering processes do not create orphan states. Figure 2 illustrates how this might happen when recovering a single process. In the figure, P_1, upon restart, requests all determinants from P_2. P_2 replies with its determinant set. However, it does so before receiving message m', so that the information sent to P_1 does not include the ordering information for m. If P_2 then receives m' it may create an orphan state because the receive of m' will be in the run but the receive of m (and therefore the send of m') may not.

The solution is for P_1 to include, in all messages, the number of the incarnation from which a message is sent.



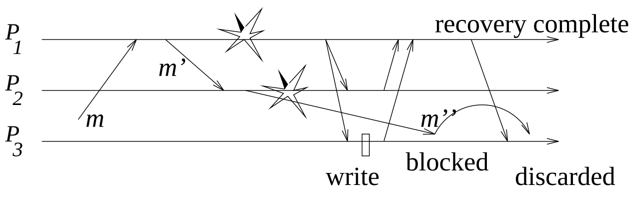
When recovery information is requested, P_1 indicates that it is entering a new incarnation, and any messages subsequently received from earlier incarnations are discarded. In the figure, P_2 would know to discard m'.

The solution for single failures does not keep orphan states from being created when there are multiple failures. Figure 3 demonstrates this. In this example, first presented in [7], P_2 fails as well. Here, even though P_1 has indicated its new incarnation number, P_3 may still create an orphan state because m'' only includes the incarnation number for P_2, but not for P_1. P_3, not knowing of P_2's failure, receives m'' and creates an orphan state.

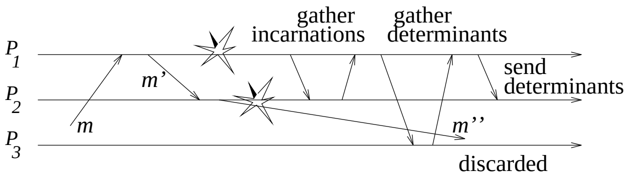


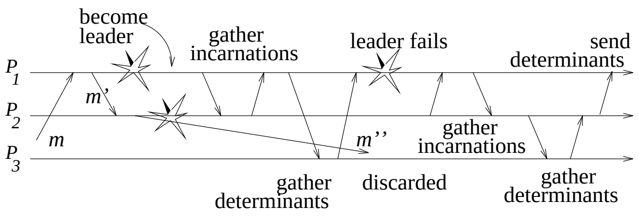
While it may seem that the solution is to have m'' include the incarnation of P_1 as well as P_2, there is a less costly solution, which we will present in the next section. First, however, we will discuss previous solutions.

Alvisi, [1], and Johnson and Zwaenepoel [9] avoid this problem using the following idea: processes that respond with determinant information are required to first write this information to stable storage. These processes must then also block the receive of any messages until after a special message has been received from the recovering process indicating recovery is complete. This completion message lists the determinants received and used for recovery. The blocking process then knows which messages can be received and which must be discarded. Figure 4 shows this scenario. In the figure, P_3 writes its recovery reply information to stable storage before replying. It then blocks the receive of m'' until the recovery complete message is received from P_1. When this message is received, m is not part of P_1's execution, and so m'' is discarded since it depends on m. The drawbacks are clear: this approach delays execution of the entire computation; every process must block while writing to stable storage and then while waiting for the failed process to recover.



Another solution was presented by Elnozahy [7]. In this solution, a process must determine when a failure occurred relative to all others. Therefore, it numbers its failure as the ordth failure in the system. For all processes that are recovering, the process with the lowest ord becomes the leader. The leader then requests the incarnation number from all recovering processes. From these it creates an incarnation vector and sends it to all live processes. The live processes use the vector to discard messages that originated from earlier incarnations. The live processes respond by sending all determinant information to the leader. The leader then forwards this to all recovering processes. This is shown in Figure 5. If a failure of a live process occurs before sending a response with determinant information, the leader starts over. If the leader fails, the next process becomes leader. Figure 6 shows this.

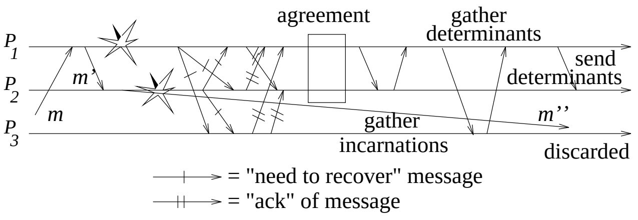




There are several problems with this approach:

- It requires knowledge of recovering processes. This knowledge would have to be manifested as a broadcast message from a process that begins to recover. The broadcast would indicate that it is part of the recovering set.
- It requires knowledge of failure order.

Since recovery detection and ordering are not free, messages must be exchanged to perform these tasks. Suppose a process, upon recovering, performs recovery detection by sending "need to recover" messages. In response, all processes acknowledge with a message that indicates whether they are recovering or not. If two processes fail simultaneously, they must also agree as to the order of their failure. Figure 7 shows what the recovery might be with this approach.



3. Our solution

For handling single failures, we employ the same technique as in [1, 7, 9]. That is, we require processes to include their incarnation number with every message sent. The single failure case also requires that every process maintain the greatest incarnation number known for every other process. We call this the *incarnation vector*. We use f (a failure count) in our algorithm to represent this vector and $s.\mathsf{f}$ to represent the value of f at state s.

For handling multiple failures (the scenario in figure 3), we require that a recovering process accept determinant information from a process only when that process's incarnation vector is greater than or equal to its own. As we will show later, this requirement allows us to satisfy the property of "no orphan states."

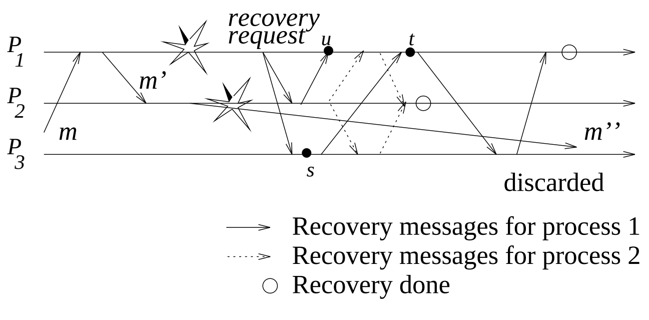
Figure 8 gives our algorithm which implements our requirements for handling single and multiple failures. We briefly describe it here. In the algorithm, f_m is an incarnation vector received in a message and *recovering* is a boolean variable. When a failure occurs, the recovering process reads, increments and then writes its incarnation number to stable storage as in the other approaches. Then it sends its incarnation number to all processes as a *recovery request* message (R1). Upon receipt of a recovery request message a process updates its incarnation vector. The process then sends its incarnation vector with determinant information to the recovering process (R2). The recovering process receives these *recovery reply* messages, and if all incarnation vectors agree, the recovering process is done (R4 and R5). However, if a reply message is received with an incarnation entry that is less than that held by the recovering process, the reply is discarded and a new request is made which includes

```
(R1) upon restart after failure do
        read f from stable storage;
        increment f[k] and write it to stable storage;
        recovering = TRUE;
        send f[k] with recover request to all processes;
(R2) upon receiving recover request from $P_j$ do
        for all $l$: f[$l$] $=$ max(f[$l$],f$_m[l]$));
        send f and determinants to $P_j$ as recovery reply;
(R3) upon receiving recovery reply from $P_j$ do
        if ($\exists l$ :f$_m[l] <$ f[$l$]) then
            send f with recover request to $P_j$;
                /* discard recovery reply. */
        else if ($\exists l$ :f$_m[l] >$ f[$l$]) then
            f = f$_m$;
            send f with recover request to all
              processes that have already replied;
(R4)  else
            update determinant information
(R5)          if determinant information updated from
              all processes then
                recreate messages;
                recovering = FALSE;
                write f to stable storage;
```

P_k

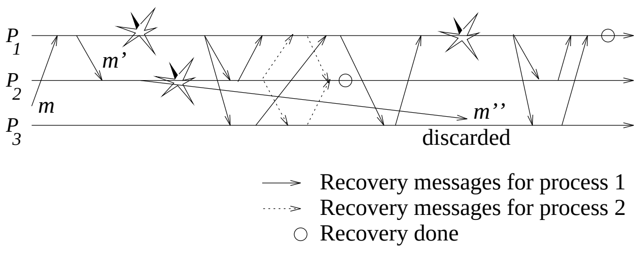
the latest incarnation vector. If a reply is received with an incarnation entry greater than what has already been received, the reply is accepted and a new request is sent to all earlier received replies (R3).

Figure 9 shows how the receipt of m'' (from figure 3 is avoided. In the figure, suppose both P_1 and P_2 have incarnation numbers of 2 when they start recovery. At s P_3 would believe P_2's incarnation number to be less than 2. When P_1 receives P_3's recovery reply at t, it immediately requests a new recovery reply because P_1 discovered P_2's failure at u. Figure 10 demonstrates that the repeated failure of one process does not necessarily affect the recovery of another process. In this figure, P_2 recovers despite P_1's failure.



recovery
request
u
t
P1
P2
P3
m'
m
m''
s
discarded
Recovery messages for process 1
Recovery messages for process 2
Recovery done

In Elnozahy's approach any failure during recovery affects all processes trying to recover. Therefore, our approach has the following benefits:



P1
P2
P3
m'
m
m''
discarded
Recovery messages for process 1
Recovery messages for process 2
Recovery done

- Knowledge of concurrent failures discovered automatically.
- No ordering of failures is required, all are independent.

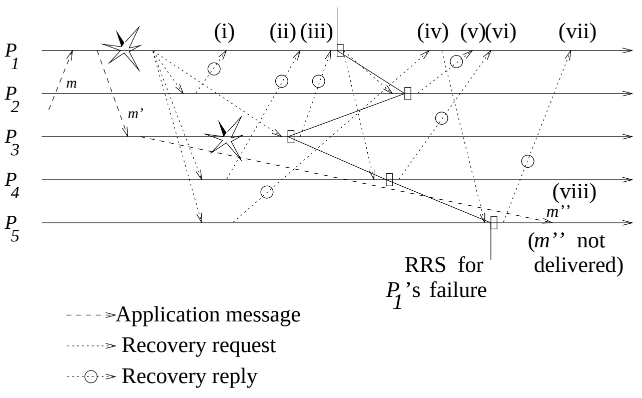
The basic idea behind our approach and that of Elnozahy is that all processes that respond with determinant information must do so with knowledge of which incarnation the other replies are coming from. We observe that these replies must create a global state for which every local state has the same incarnation information. We call this global state a *recovery reply state*. This state need not be consistent (that is, receive states may precede the *recovery reply state*, without their corresponding send states doing the same [5]). We will prove the existence of such a state for all failures when proving correctness.

Definition 3.1 *The* **recovery reply state (RRS)** *for failure x at P_k is the least global state, G, across all processes such that:*

1. *The recovery state (for failure x) $\preceq G[k]$*
2. *The* recovering *variable at $G[k]$ equals TRUE*
3. *For all i: $G[i]$.f = $G[k]$.f = f at (R5) for P_k.*

An RRS acts as a barrier to messages sent before earlier failures. No message sent before an RRS with an incarnation number less than f for the RRS will be delivered after the RRS. Our recovery algorithm ensures that recovery reply messages are all received with the same view of incarnations (that is, that an RRS is created).

We will show in the next section that ensuring an RRS exists for every recovery, maintains the property of "no orphan states." Figure 11, although complex, gives an example of the creation of an RRS for a failure at P_1 when it occurs near in time to another failure. The recovery replies received at i, ii, and iv are invalid (and in fact the reply at iv is rejected) since P_3 has also failed and increments its incarnation number so that it doesn't match those of the previous replies. (This reply from P_3 is received at iii.) Therefore, P_1 updates f and requests determinants from P_2 and P_4 again and



recovery reply stateP_1

from P_5 when its reply is later received at iv. P_1 receives the second replies at v, vi, vii. P_1 also reaches R5 in our algorithm at vii. The solid line across the computation in the figure therefore represents the recovery reply state for P_1's failure.

4. Correctness

We now demonstrate that our algorithm maintains the property of "no orphan states." We assume that given determinant information, a process can recreate and replay messages. This can be done by requesting that processes resend messages. It is not too difficult to show that this is always possible (see [1]).

We first show that for every failure a recovery reply state exists.

Lemma 4.1 *For every failure x, there exists a recovery reply state (RRS).*

Proof: Let x be a failure on P_k. When P_k fails we assume that it is restarted. Let $\mathcal{G}$ be the set of global states that satisfy conditions 1,2, and 3 of the definition of RRS (Definition 3.1). We prove first that $\mathcal{G}$ is not empty and second that there exists a least global state in $\mathcal{G}$.

$\mathcal{G}$ **is not empty:** Upon restarting following failure x, P_k executes lines R1. The state, s, at which P_k completes R1 satisfies conditions 1 and 2. That is, its recovery state $\preceq s$ and recovering = TRUE at s. Also, f has been updated and sent to all processes in P_k's group. P_k will eventually receive a *recovery reply* message (it can rerequest a reply in case it believes another process has failed or that a message was lost). If no other failures have occurred, all replies received will contain an incarnation vector equivalent to s.f, and $\forall i \neq k$: $G[i]$ is the state following the receive of the *recovery request*. If other failures have occurred f received by P_k will not equal s.f. By lines R3, P_k updates its f and resends *recover request* messages until all *recovery reply* messages received contain an f equal to its own. Since we assume that processes always eventually respond, and that every process fails a finite number of times, P_k will eventually gather all replies necessary. If the final *recovery request* message is sent at s'', by previous assumptions all processes will eventually have a state at which f = s''.f. Let t be the first state in which f at P_k = s''.f. Now since $s \preceq t$, condition 1 is satisfied for $G[k] = t$. Condition 2 is satisfied because t will precede the state at which the final correct reply is received. For every other process i, $G[i]$ is the first state for which f = s''.f, which satisfies condition 3. Therefore there is a G that satisfies conditions 1, 2, and 3.

There is an earliest G**:** We know that the earliest value agreed to is t.f (P_k stops sending *recovery request* messages when all replies equal its own f by lines R3.) We know that there is a G in which $\forall i \neq k$: $G[i]$.f = t.f. Since there must be an earliest state at every P_i which sets f = t.f, let $G[i]$ be this state. Therefore, there is an earliest G. □

We will show in the next lemma that messages such as m'' in Figure 11 (for which there is a failure that follows the send on the same process and precedes the recovery reply state) will not be delivered after the recovery reply state.

Lemma 4.2 *No message will be received following a recovery reply state, RRS, if there has been an intervening failure between its send and the RRS.*

Proof: Let $m.send$ and $m.receive$ represent the send and receive states for a message m. We can state the lemma as follows:
$\forall m$ sent from P_j to P_k, where m is an application message:
$(\exists x : x =$failure state: $m.send \prec x \prec RRS[j]) \Rightarrow (RRS[k] \not\prec m.receive)$.

Note first that if $\exists x : x =$failure state: $m.send \prec x \prec RRS[j]$ then

$$m.send.\mathsf{f}[j] < RRS[j].\mathsf{f}[j]. \qquad (i)$$

This is because when P_j recovers after x, it performs R1 before anything else. Therefore, any state after x will have f$[j]$ greater than f$[j]$ from any state before x.

From definition of RRS,

$$\forall l, RRS[l].\mathsf{f} = RRS[i].\mathsf{f}. \qquad (ii)$$

(i) and (ii) imply $m.send.\mathsf{f}[j] < RRS[k].\mathsf{f}[j]$. Because of this, m will not be received for any state which follows RRS$[k]$. □

Now we can show that our algorithm does not allow orphan states.

Theorem 4.3 *The recovery algorithm of figure 8 does not create orphan states.*

Proof: We must show that:

$$\forall s, t : s \notin \text{run} \Rightarrow t \notin \text{run} \lor s \not\to t$$

Equivalently, if a send state is not in a run, then neither is the corresponding receive state.

A send state, s, is not in a run if a failure occurs following s and all determinants for receives which precede s are not recovered in some subsequent recovery. That is, s is not part of the run if $s \prec$ failure state x and the determinants at s are not part of some recovery which follows x.

Let s's corresponding receive state, be t. Let y be a failure state whose recovery reply state (RRS_y) follows x (x may equal y). If s is not in the run, then

$$\forall k : t \not\prec RRS_y[k] \quad (*).$$

That is, t must follow some recovery so that the determinants at s are not part of that recovery. (There are other requirements as well, however, we don't need these for our proof).

If (*) holds then recovery of y may not contain the determinants necessary to recreate s. That is, (*) means that $\exists j : s \prec x \prec RRS_y[j]$. To prove our theorem we must show that t is not in the run. Since $\forall k : t \not\prec RRS_y[k]$ and by the previous lemma, t will not be received following RRS_y (because $\exists j : s \prec x \prec RRS_y[j]$), t is not part of the run. $\Box$

5. Conclusions

We have presented an approach for recovery in causal message logging which does not require unfailed processes to block or require special coordination among recovering processes.

6. Acknowledgements

We would like to thank Lorenzo Alvisi for valuable discussions concerning causal message logging.

References

[1] L. Alvisi. "Understanding the message logging paradigm for masking process crashes." In *Ph.D. dissertation*. Cornell University, January 1996.

[2] L. Alvisi, K. Marzullo, "Trade-Offs in Implementing Optimal Message Logging Protocols," *Proceedings of the 15th ACM Symposium on the Principles of Distributed Computing*, ACM, May 1996.

[3] L. Alvisi, S. Rao, H. Vin, "Understanding Recovery in Causal Message Logging," Computer Science Department, Technical report, University of Texas at Austin (in preparation).

[4] A. Borg, J. Baumbach, S. Glazer, "A message system supporting fault tolerance," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, ACM, October 1983, pp. 90-99.

[5] K. Chandy, L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, ACM, February 1985, pp. 63-75.

[6] E. Elnozahy, W. Zwaenepoel, "Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit," *IEEE Transactions on Computers*, Vol. C-41, No. 5, May 1992, pp. 526-531.

[7] E. Elnozahy, "On the relevance of Communication Costs of Rollback-Recovery Protocols," *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, 1995, pp.74-79.

[8] D. Johnson and W. Zwaenepoel. "Sender-based message logging." *17th Annual International Symposium on Fault-Tolerant Computing*, pages 14–19, June 1987.

[9] D. Johnson, W. Zwaenepoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," *Journal of Algorithms*, Vol. 11, Sept. 1990, pp.462-491.

[10] D. Johnson, "Efficient Transparent Optimistic Rollback Recovery for Distributed Application Programs," *Proceedings of the 12th Symposium on Reliable Distributed Systems*, IEEE Computer Society, October 1993, pp.86-95.

[11] M. L. Powell, D. L. Presotto, "Publishing: a reliable broadcast communication mechanism," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, ACM, October 1983, pp. 100-109.

[12] A.P. Sistla, J.L. Welch, "Efficient Distributed Recovery Procedure Using Message Logging," *Eighth ACM Symposium on Principles of Distributed Computing*, 1989, pp.223-238.

[13] S.W. Smith, D.B. Johnson, J.D. Tygar, "Completely asynchronous recovery with minimal rollbacks," *Proceedings of the 25th International Symposium on Fault Tolerant Computing*, 1995, pp.361-370.

[14] R. Strom, D. Bacon, and S. Yemeni. "Volatile logging in n-fault-tolerant distributed systems." *Proceedings fo the 18th International Symposium on Fault-Tolerant Computing*, pages 44–49, June 1988.

[15] R. Strom, S. Yemeni, “Optimistic Recovery in Distributed Systems,” *ACM Transactions on Computer Systems*, Vol. 3, August 1985, pp.204-226.