How to Recover Efficiently and Asynchronously when Optimism Fails

Om P. Damani   *Vijay K. Garg*

TR TR-PDS-1995-014 August 1995

Parallel & Distributed Systems group
Department of Electrical & Computer Engineering
University of Texas at Austin
Austin, Texas 78712

How to Recover Efficiently and Asynchronously when Optimism Fails

Om P. Damani
damani@cs.utexas.edu
Dept. of Computer Sciences

Vijay K. Garg*
garg@ece.utexas.edu
Dept. of Electrical and Computer Engineering

University of Texas at Austin, Austin, TX, 78712

August 14, 1995

Abstract

We propose a new algorithm for recovering asynchronously from failures in a distributed computation. Our algorithm is based on two novel concepts - a *fault-tolerant vector clock* to maintain causality information in spite of failures, and a *history* mechanism to detect orphan states and obsolete messages. These two mechanisms together with checkpointing and message-logging are used to restore the system to a consistent state after a failure of one or more processes. Our algorithm is completely asynchronous. It handles multiple failures and network partitioning, does not assume any message ordering, causes the minimum amount of rollback and restores the maximum recoverable state with low overhead. Earlier optimistic protocols lack one or more of the above properties.

1 Introduction

For fault-resilience, a process periodically records its state on a stable storage [15]. This action is called checkpointing and the recorded state is called a checkpoint. The checkpoint is used to restore a process after a failure. However, some information may be lost in restoring the system. This loss may leave the distributed system in an *inconsistent* state [5]. The goal of a recovery protocol is to bring back the system to a *consistent* state after one or more processes fail. A *consistent* state is one where the send of a message must be recorded in the sender's state if the receipt of the message has been recorded in the receiver's state.

In *consistent checkpointing*, different processes synchronize their checkpointing actions [4, 13]. After a process fails, some or all of the processes rollback to their last checkpoints such that the resulting system state is consistent. For large systems, the cost of this synchronization is prohibitive. Furthermore, these protocols may not restore the maximum recoverable state [12].

If along with checkpoints, messages are logged to the stable storage, then the maximum recoverable state can always be restored [12]. Theoretically, message logging alone is sufficient, but

*supported in part by the NSF Grant CCR-9110605, a TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant.

checkpointing speeds up the recovery. Messages can be logged either by the sender or by the receiver. In *pessimistic* logging, messages are logged either as soon as they are received [3, 20], or before the receiver sends a new message [11]. When a process fails, its last checkpoint is restored and the logged messages that were received after the checkpointed state are replayed in the order they were received. Pessimism in logging ensures that no other process needs to be rolled back. Although this recovery mechanism is simple, it reduces the speed of the computation. Therefore, it is not a desirable scheme in an environment where failures are rare and message activity is high.

In *optimistic* logging [12, 19, 25, 26, 27], it is assumed that failures are rare. A process stores the received messages in volatile memory and logs it to stable storage at infrequent intervals. Since volatile memory is lost in a failure, some of the messages can not be replayed after the failure. Thus, some of the process states are *lost* in the failure. States in other processes that depend on these lost states become *orphan*. A recovery protocol must rollback these orphan states to *non-orphan* states. The following properties are desirable for an optimistic recovery protocol:

- *Asynchronous recovery*: A process should be able to restart immediately after a failure [25, 27]. It should not have to wait for messages from other processes.

- *Minimal amount of rollback*: In some algorithms, processes which causally depend on the lost computation might rollback more than once. In the worst case, they may rollback an exponential number of times. This is called the *domino* effect [21, 22]. A process should rollback at most once in response to each failure.

- *No assumptions about the ordering of messages*: If assumptions are made about the ordering of messages such as FIFO, then we lose the asynchronous character of the computation [19]. A recovery protocol should make as weak assumptions as possible about the ordering of messages.

- *Handle concurrent failures*: It is possible that more than one processes fail concurrently in a distributed computation. A recovery protocol should handle this situation correctly and efficiently [25, 27].

- *Low overhead*: The algorithm should have a low overhead in terms of number of control messages or the amount of control information piggybacked on application messages, both during a failure-free operation and during recovery.

- *Tolerate network partitioning*: A process should not depend upon information stored in other processes to recover. It should be able to restart despite network partitioning [25, 27].

- *Recover maximum recoverable state*: No computation should be needlessly rolled back.

We present an optimistic recovery protocol which has all the above features. Previous protocols lack one or more of these properties. Table 1 shows a comparison of our work with some other schemes. Our protocol is based on two novel mechanisms - a *fault-tolerant vector clock* and a *history* mechanism. The fault-tolerant vector clock is used to maintain causality information in spite of failures. This mechanism is of independent interest as it can also be applied to other distributed algorithms such as distributed predicate detection [9]. The history mechanism is used to detect orphan states and obsolete messages. In this paper, we present necessary and sufficient conditions for a message to be obsolete and a state to be orphan in terms of the history data structure.

The organization of the rest of the paper is as follows. Section 2 discusses the related work in the literature. In Section 3, we discuss our model of computation. In particular, we extend Lamport's 'happen before' relation which has been quite useful for ordering events in a failure-free system to a system where processes fail and rollback. Section 4 presents an algorithm to maintain Fault-Tolerant Vector Clocks. It also shows how to use them for detecting 'happen before' relation between states that are neither lost nor orphan. Section 5 gives an algorithm for history maintenance using which orphan states are detected and rolled back. Section 6 presents and analyzes our protocol. Section 7 concludes the paper.

	Message ordering	Asynchronous recovery	Maximum rollbacks per failure	Number of timestamps in vector clock	Number of concurrent failures allowed
Strom and Yemini [27]	FIFO	Yes	$\Theta(2^n)$	$O(n)$	1
Johnson and Zwaenepoel [11]	None	No	1	$O(1)$	n
Sistla and Welch [26]	FIFO	No	1	$O(n)$	1
Peterson and Kearns [19]	FIFO	No	1	$O(n)$	1
Smith, Johnson and Tygar [25]	None	Yes	1	$O(n^2 f)$	n
Damani and Garg	None	Yes	1	$O(n)$	n

Table 1: Comparison with related work. (n is the number of processes in the system and f is the maximum number of failures of any single process).

2 Related Work

A protocol for recording a consistent global state was first given by Chandy and Lamport [5]. Although it did not deal with recovery, much of the earlier work on recovery is based on it. Strom and Yemini [27] initiated the area of optimistic recovery using checkpointing. Their scheme, however, suffers from the *domino* effect. Johnson and Zwaenepoel [12] present a centralized protocol to optimistically recover the maximum recoverable state. Other distributed protocols for optimistic recovery can be found in [19, 25, 26]. Peterson and Kearns [19] give a synchronous protocol based on vector clock. Their protocol can not handle multiple failures. Smith, Johnson and Tygar [25] present the first completely asynchronous, optimistic protocol which can handle multiple failures. They maintain information about two levels of partial order: one for the application and the other for the recovery. The main drawback of their algorithm is the size of its vector clock, resulting in high overhead during failure-free operations. An optimistic protocol for *fast output* to *environment* is presented in [10].

Pessimistic protocols can be found in [3, 11, 20]. *Causal logging* [1, 6] protocols are non-blocking and orphan free. They log message in processes other than the receiver. So synchronization is required during recovery. Alvisi and Marzullo [2] present a theoretical framework for different message logging protocols. Leong and Agrawal [16] take message semantics into account to reduce rollback. Recovery algorithms for Distributed Shared Memory are given in [18, 23, 24]. By using the technique presented in the [7], recovery algorithms for message passing architecture can be extended to Distributed Shared Memory.

3 Our Model of Computation

A distributed computation is a set of process executions. A process execution is a sequence of states in which a state transition is caused by an external event: a send or a receive of a message. Internal events do not cause state transitions; we ignore them for the rest of the paper. Processes are assumed to be piecewise deterministic. This means that when a process receives a message, it performs some internal computation, sends some messages and then blocks itself to receive a message. All these actions are completely deterministic, i.e. actions performed after a message receive and before blocking for another message receive are completely determined by the contents of the message received and the state of the process at the time of message receive. A non-deterministic action can be modeled by treating it as a message receive.

The receiver of a message depends on the content of the message and therefore on the sender of the message. This dependency relation is transitive. The receiver becomes dependent only after the received message is delivered. From now on, unless otherwise stated, receive of a message will imply its delivery.

A process periodically takes its checkpoint. It also asynchronously logs to the stable storage all messages received in the order they are received. At the time of checkpointing, all unlogged messages are also logged.

A failed process *restarts* by creating a new version of itself. It restores its last checkpoint and replays the logged messages which were received after the restored state. Since some of the messages might not have been logged at the time of the failure, some of the old states, called *lost* states, can not be recreated. Now, consider the states in other processes which depend on the lost states. These states, called *orphan* states, must be rolled back. Other processes have not failed, so before rolling back, they can log all the unlogged messages and save their states. Thus no information is lost in rollback. Note the distinction between restart and rollback. A failed process restarts whereas an orphan process rolls back. Some information is lost in restart but not in rollback. A process creates a new version of itself on restart but not on rollback. A message sent by a lost or an orphan state is called an *obsolete* message. A process receiving an *obsolete* message must discard it. Otherwise the receiver becomes an *orphan*.

In Figure 1, a distributed computation is shown. Process P1 fails at state f10, restores state s11, takes some actions needed for recovery and restarts from state r10. States s12 and f10 are lost . Being dependent on s12, state s22 of P2 is an *orphan*. P2 rolls back, restores state s21, takes actions needed for recovery, and restarts from state r20. Dashed lines show the lost computation. Solid lines show the useful computation at the current point.

Henceforth, notation i, j refer to process numbers; k, l, v refer to version number of a process;

s, u, w, x, y refer to a state; P_i refers to process i; $P_{i,k}$ refers to version k of P_i; $s.p$ denotes the process number to which s belongs, that is, $s.p = i \Rightarrow s \in P_i$; t, t', t'' refer to timestamp; m refers to a message.

Lamport [14] defined the *happen before* relation between events in a failure-free computation. To take failures into account, we extend the *happen before*($\rightarrow$) relation. We define it for states. For the states s and u, $s \rightarrow u$ is the transitive closure of the relation defined by the following three conditions:

- $s.p = u.p$ and s was executed immediately before u (for example, $s11 \rightarrow s12$ in Figure 1), or
- $s.p = u.p$ and s is the state restored after a failure or a rollback and u is the state after $P_{u.p}$ has taken the actions needed for recovery (for example, $s11 \rightarrow r10$ in Figure 1), or
- s is the sender of a message m and u is the receiver of m (for example, $s00 \rightarrow s11$ in Figure 1).

In figure 1, s00 $\rightarrow$ s22, but s22 $\not\rightarrow$ r20 (*not happen before*).

The protocol for recovery might cause some recovery messages to be sent among processes. ¿From here onward 'application message' will be referred to as 'message' and 'recovery message' will be referred to as 'token'. Tokens do not contribute to *happen before*; if s sends a token to u then because of this token, s does not become causally dependent on u.

We say that s *knows* about $P_{i,l}$ through token or messgaes if,

1. $\exists u : u.p = s.p$ and u has received a token about $P_{i,l}$ and u was executed before s, or,
2. $\exists u : u \rightarrow s$ and $u \in P_{i,l}$.

4 Fault-Tolerant Vector Clock

Mattern's vector clock [17] is a vector whose number of component equals the number of processes. Each entry is the timestamp of the corresponding process. To maintain causality despite failures, we extend each entry by a *version number*. The extended vector clock is referred to as the *Fault-Tolerant Vector Clock* (FTVC). We use the terms 'clock' and FTVC interchangeably. Let us consider the FTVC of a process P_i. The version number in the i'th entry of its FTVC (its own version number) is equal to the number of times it has failed and recovered. The version number in the j'th entry is equal to the highest version number of P_j on which P_i depends. Let entry e corresponds to a tuple(version v, timestamp ts). Then, $e_1 < e_2 \equiv (v_1 < v_2) \vee [(v_1 = v_2) \wedge (ts_1 < ts_2)]$.

A process P_i sends its FTVC along with every outgoing message. After sending a message, P_i increments its timestamp. On receiving a message, it checks whether the message is obsolete or not (we will explain later how to do this). If the message is obsolete it is discarded; otherwise, the process updates its FTVC with the message's FTVC by taking the componentwise maximum of entries and incrementing its own timestamp. To take the maximum, the entry with the higher version number is chosen. If both entries have the same version number then the entry with the higher timestamp value is chosen.

When a process restarts after a failure, it increments its version number and sets its timestamp to zero. Note that this operation does not require access to previous timestamp which may be lost

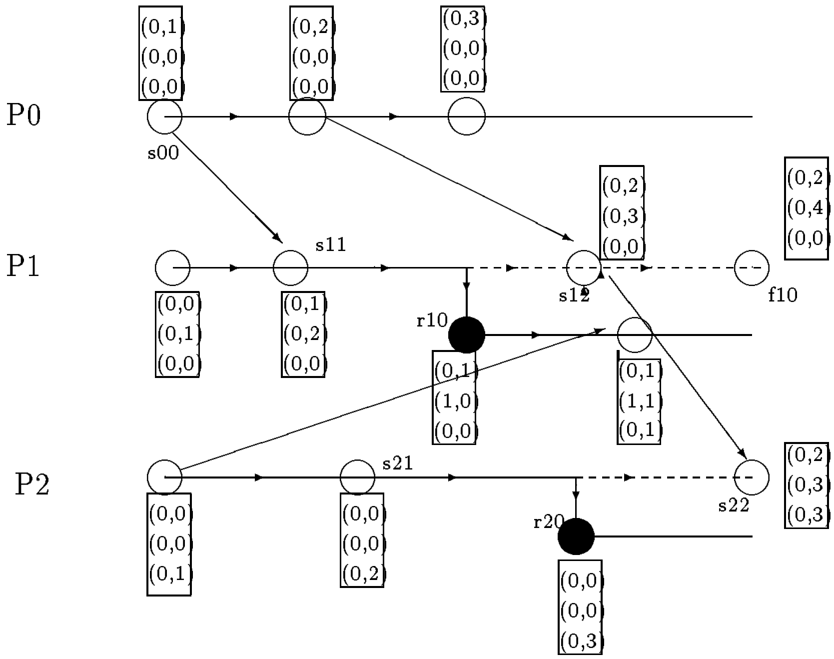

Figure 1: A Distributed Computation

on a failure. It only requires its previous version number. As explained in section 6.2, the version number is not lost in a failure.

After a rollback, a process increments the timestamp of its own component and leaves the version number unchanged. A formal description of the FTVC algorithm is given in Figure 2.

An example of FTVC is shown in Figure 1. FTVC of each state is shown in a rectangular box near it.

4.1 Properties of FTVC

FTVC has properties similar to Mattern's vector clock [17]. It can be used to detect causal dependencies between *useful* states, that is, the states which are neither lost nor orphan.

We define ordering between two FTVC values $c1$ and $c2$ as follows.

$$c1 < c2 \equiv (\forall i : c1[i] \leq c2[i]) \wedge (\exists j : c1[j] < c2[j]).$$

Let $s.c$ denote the FTVC of $P_{s.p}$ in state s. The following lemma formalizes the meaning of a version number in an entry of FTVC.

Lemma 1 *Let $s \in P_i$. Then,*

1. $s.c[i].ver = |\{u : u \in P_i, u \text{ is a failure event }\}|$
2. $(j \neq i), s.c[j].ver = max_{u \in P_j}\{u.c[j].ver : u \to s\}$

Proof: On restarting after a failure, a process increments its version number. Furthermore, it increments its version number only then. Hence the first part follows.

The relation $u \to s$ implies that there exists a message path between u and s. Now $c[j].ver$ can only be incremented by P_j. Since u is the maximum state in P_j, along each link of the path

```
Process P_i :

type entry =      (int ver, int ts)          /* version, timestamp */
var clock :       array [1..N] of entry      /* N : number of processes in system */

Initialize :
    ∀ j : clock[j].ver = 0 ; clock[j].ts = 0 ;
    clock[i].ts = 1 ;
Send_message :
    send (data, clock) ;
    clock[i].ts++ ;
Receive_message (data, mclock) :
    /* P_i receives vector clock 'mclock' in incoming message */
    ∀ j: clock[j] = max(clock[j],mclock[j]) ;
    clock[i].ts++ ;
On Restart(state s restored after failure ) :
    /* clock = s.clock */
    clock[i].ver++ ;
    clock[i].ts = 0 ;
On Rollback (state s is restored) :
    /* clock = s.clock */
    clock[i].ts++ ;
```

Figure 2: Formal description of the fault-tolerant vector clock

from u to s, FTVC$[j].ver$ is updated by taking componentwise maximum only. So, $\forall u : (u \to s) \Rightarrow s.c[j].ver \geq u.c[j].ver$. Or, $s.c[j].ver \geq max_{u \in P_j}\{u.c[j].ver | u \to s\}$. Since $s.c[j].ver$ is updated by taking a maximum only and it is never incremented otherwise, $s.c[j].ver \leq max_{u \in P_j}\{u.c[j].ver | u \to s\}$. Hence the second part follows. ■

The next lemma gives a necessary condition for '$\not\to$' relation between two *useful* states.

Lemma 2 *Let s and u be useful states (neither lost nor orphan) and $s \neq u$. Then, $s \not\to u \Rightarrow u.c[s.p] < s.c[s.p]$*

Proof: Let $s.p = u.p$. Since s and u are useful states it follows that $u \to s$. After send and receive of a message or a rollback, $P_{s.p}$ increments the timestamp of its own component. On restart after a failure, $P_{s.p}$ increments its version number. Since for each state transition along the path from u to s, local FTVC is incremented, $u.c[s.p] < s.c[s.p]$.

Let $s.p \neq u.p$. As $s \not\to u$, $P_{u.p}$ could not have seen $s.c[s.p]$, local clock of $P_{s.p}$. This argument can be formalized using the induction technique on '$\not\to$', given by Garg and Tomlinson [8]. Hence $u.c[s.p] < s.c[s.p]$. ■

As shown in the next theorem, the above condition is also sufficient for '$\not\to$' relation. The next theorem shows that despite failures, FTVC keeps track of causality for the *useful* states. This may be of interest in applications other than recovery, for example, in predicate detection.

Theorem 1 *Let s and u be useful states in a distributed computation. Then, $s \to u$* **iff** *$s.c < u.c$*

Proof: If $s = u$, then the theorem is trivially true. Let $s \to u$. There is a message path from s to u such that none of the intermediate states are either lost or orphan. Due to monotonicity of the FTVC along each link in the path, $\forall j : s.c[j] \leq u.c[j]$. Since $u \not\to s$, from lemma 2, $s.c[u.p] < u.c[u.p]$. The converse follows from lemma 2. ■

Note that the FTVC does not detect the causality for either lost or orphan states. In Figure 1, r20.$c <$ s22.c, even though r20 $\not\to$ s22. To detect causality for lost or orphan states, we use *history*, as explained in Section 5.

5 History Mechanism

We first give some definitions which are similar to those in [19]. A state is called *lost*, if it cannot be restored from the stable storage after a process fails. To define a lost state more formally, let $restored(u)$ denote the state that is restored after a failure. Then,

$$lost(s) \equiv \exists u : restored(u) \wedge u.p = s.p \wedge u.ver = s.ver \wedge u \to s$$

That is, a state s is lost if there exists a state u which was restored after a failure and s was executed after u in that version of the process.

States in other processes which are dependent on a lost state are called *orphan*. Formally,

$$orphan(s) \equiv \exists u : lost(u) \wedge u.p \neq s.p \wedge u \to s$$

A message sent by a lost or an orphan state is not useful in the computation and it should be discarded. It is called *obsolete*. Formally,

$$obsolete(m) \equiv lost(m.sender) \vee orphan(m.sender)$$

If an obsolete message has been received then the receiver should rollback.

Orphan states and resulting obsolete messages are detected using the history mechanism. This method requires that after recovering from a failure, a process notifies other processes by broadcasting a *token*. The token contains the version number which failed and the timestamp of that version at the point of restoration. We do not make any assumption about the ordering of tokens among themselves or with respect to the messages. We do assume that tokens are delivered reliably.

Every process maintains some information, called history, about other processes in its volatile memory. In history of P_i, there is a record for every *known* version of all processes. If P_i has received a token about $P_{j,k}$, then it keeps that token's timestamp in the corresponding record in history. Otherwise, it keeps the highest value of the timestamp that it *knows* for $P_{j,k}$ through messages. A bit is kept to indicate whether the stored timestamp corresponds to a token or a message. So a record in history has three fields: a bit, a version number and a timestamp. The routine insert(history[j], hist_entry) inserts the record *hist_entry* in that part of the history of P_i which keeps track of P_j. For a given version of a given process, only one record is maintained. So on adding a record for $P_{j,v}$, any previous record for $P_{j,v}$ is deleted. Thus, on receiving a message and its FTVC, for each entry $e_j(v,t)$ in the vector clock, P_i checks whether a record (mes, v, t') exists for $P_{j,v}$ in history$[j]$ such that $t < t'$. If no such record exists then record (mes, v, t) is added to history$[j]$. By adding this record, any previous record for $P_{j,v}$ is deleted.

A formal description of the history manipulation algorithm is given in Figure 3.

```
Process P_i :

type entry =        (int ver, int ts)          /* version, timestamp */
     hist_entry =   record ( mtype : (token,mes), int ver, int ts)
var  clock :        array[1..N] of entry;      /* N : number of processes in system */
     history :      array[1..N] of set of hist_entry;
     token :        entry;

Initialize :
     ∀j : insert(history[j], (mes,0,0)) ;
     insert(history[i], (mes,0,1)) ;
Send_message :
     send(data, clock) ;
Receive_token (v1,t1) from P_j :
     insert(history[j], (token,v1,t1)) ;
Receive_message (data, mclock) :
     ∀ j :if ((mes,mclock[j].ver,t) ∉ history[j])    /* A record for mclock[j].ver does not exist */
             or (t < mclock[j].ts) then               /* or it exists and t is the time-stamp in it */
             insert(history[j], (mes,v,mclock[j].ts)) ;
On Restart (state s is restored after a failure of version v)
     /* history = s.history */
     insert(history[i], (token,v,clock[i].ts)) ;
```

Figure 3: A formal description of the history mechanism

6 The Protocol

Our protocol for asynchronous recovery is shown in Figure 4. We describe the actions taken by a process, say P_i, upon the occurrence of different events.

6.1 Message Receive

On receiving a message, P_i first checks whether the message is obsolete. This is done as follows. Let e_j refer to the jth entry in the message's FTVC. Recall that each entry is of the form (v, t) where v is the version number and t is the timestamp. If there exists an entry e_j, such that e_j is (v, t) and $(token, v, t')$ belongs to history[j] of P_i and $t > t'$ then the message is obsolete. This is proved later.

If the message is obsolete, then it is discarded. Otherwise, P_i checks whether the message is deliverable. The message is not deliverable if its FTVC contains a version number k for any process P_j, such that P_i has not received all the tokens of the form $P_{j,l}$ for all l less than k. In this case, the delivery of the message is postponed. Since we assume failures to be rare, this should not affect the speed of the computation.

If the message is delivered then the vector clock and the history are updated. P_i updates its FTVC with the message's FTVC as explained in Section 4. The message and its FTVC is logged in a volatile storage. Asynchronously, volatile log is flushed to the stable storage. The history is updated as explained in Section 5.

6.2 On Restart after a Failure

After a failure, P_i restores its last checkpoint from the stable storage (including the history). Then it replays all the logged messages received after the restored state, in the receipt order. To inform other processes about its failure, it broadcasts a token containing its current version number and timestamp. After that it increments its own version number and resets its own timestamp to zero. Finally, it updates its history, takes a new checkpoint and starts computing in a normal fashion. The new checkpoint is needed to avoid the loss of the current version number in another failure. Note that the recovery is unaffected by a failure during this checkpointing.

6.3 On Receiving a Token

We require all tokens to be logged synchronously. This prevents the process ¿from losing the information about the token if it fails after acting on it. Since we expect the number of failures to be small, this would incur only a small overhead.

The token enables a process to discover if it has become an orphan. To check whether it has become an orphan it proceeds as follows. Assume that it received the token (v, t) from P_j. It checks whether a record (mes, v, t') exists in its history for $P_{j,v}$, such that $t < t'$. If such a record exists, then P_i is an orphan and it needs to rollback. We prove this claim later.

If the process P_i discovers that it has become an orphan then it rolls back.

Regardless of the rollback, P_i enters the record $(token, v, t)$ in history[j]. Finally, messages that were held for this token are delivered.

Process P_i :

Receive_message (data, mclock) :

/* Check whether message is obsolete */

$\forall$ j :**if** ((token,mclock[j].ver,t) $\in$ history[j]) **and** (t $<$ mclock[j].ts) **then** discard message ;

if $\exists j, l$ s.t. $l <$ mclock[j].ver $\wedge$ P_i has not received token about $P_{j,l}$ **then**

postpone the delivery of the message till that token arrives ;

if delivered **then**

update history ; update FTVC ;

Restart (after failure) :

restore last checkpoint ;

replay all the logged messages that follow the restored state ;

broadcast token (clock[i]) ;

update history ; update FTVC ;

take checkpoint ;

continue as normal ;

Receive_token (v,t) from P_j :

synchronously log the token to the stable storage ;

if ((mes,v,t') $\in$ history[j]) **then**

if (t $<$ t') **then** Rollback ;

/* Regardless of rollback, following actions are taken */

update history ;

deliver messages that were held for this token ;

Rollback (due to token (v,t) from P_j) :

log all the unlogged messages to the stable storage ;

restore the maximum checkpoint such that

either no record (mes,v,t') $\in$ history[j] or (t' $<$ t) ..(I)

discard the checkpoints that follow ;

replay the messages logged after this checkpoint till condition (I) remains satisfied ;

discard the logged messages that follow ;

update FTVC ;

continue as normal ;

Figure 4: Our Protocol for Asynchronous Recovery

6.4 On Rollback

On a rollback due to token (v, t) from P_j, P_i first logs all the unlogged messages to the stable storage. Then it restores the maximum checkpoint s such that the history of s satisfies one of the following conditions :

1. There is no record for $P_{j,v}$ in the history of s, or,
2. There is a record (mes, v, t'') for $P_{j,v}$ in the history and $t'' < t$.

These conditions imply that s is non-orphan. Then, logged messages that were received after s are replayed as long as one of the above conditions remain satisfied. It discards the checkpoints and logged messages that follow this state. Now, the FTVC is updated by incrementing its timestamp. Note that it does not increments its version number. P_i, then restarts computing as normal.

6.5 Remark

The following issues are relevant to all the optimistic protocols including ours. We just mention them and do not discuss them any further.

1. On a failure, a process loses information about the messages that it received but did not log before the failure. These messages are lost forever, unless P_i also broadcasts its clock with the token and other processes resend all the messages that they sent to P_i (only those messages need to be retransmitted whose send states were concurrent with token's state). This means that processes have to keep send-history. Observe that no retransmission of messages is required during rollback of a process which has not failed, but has become orphan due to a failure of some other process. Before rolling back, it can log all the messages and so no message is lost.

2. Some form of garbage collection is also required for reclaiming space. Space required for checkpoints and message logs can be bounded by using the scheme presented in [28]. Before committing an output to the environment, a process must make sure that it will never rollback the current state or lose it in a failure.

6.6 An Example

In Figure 5, ci is the checkpoint of process Pi. The value of the FTVC and the history is also shown for some of the states. The FTVC is shown in a box. The row i of the FTVC and the history corresponds to Pi. Some of the state transitions are not shown to avoid cluttering of the figure. The process $P1$ fails in state $f10$. It restores the checkpoint $c1$ and replays the logged messages. Then it sends the token (0,3) (shown by dotted arrow) to other processes. It restarts in state $r10$. $P0$ receives the message $m2$ in state $s03$. $m2$'s FTVC contains an entry for version 1 of $P1$. As $P0$'s history does not contain the token about version 0 of $P1$, it postpones the delivery of $m2$. It receives the token in state $s05$. It detects that it is an orphan and rolls back. It restores the checkpoint $c0$, replays the logged messages untill the message that made it an orphan. It restarts in state $r00$. Since message $m2$ was held for this token, it is delivered now. On receiving message $m0$, P2 detects that it is obsolete and discards it.

Note that if state $s03$ of $P0$ had delivered the message $m2$, then message $m0$'s FTVC would have contained entry (1,1) for $P1$. Then $P2$ would not have been able to detect that $m0$ is obsolete.

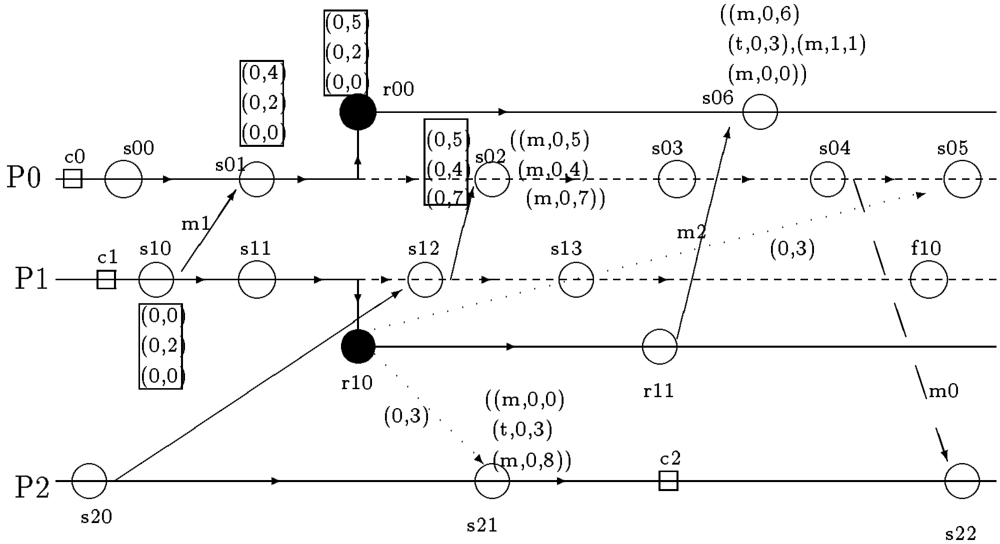

Figure 5: An example of recovery

So $P2$ would have delivered $m0$, resulting in an orphan state. Since $P2$ had already received the token for version 0 of $P1$, $P2$ would never have rolled back the orphan state.

6.7 Proof of Correctness

The following lemma gives a necessary and sufficient condition for orphan detection. This condition is used in the *Receive_token* part of the algorithm.

Lemma 3 orphan(s) $\equiv \exists w : restored(w) \wedge (w.clock = (v,t) \wedge \exists(mes, v, t') \in s.history[w.p]$ *such that* $t < t'$.

Proof: $(\Leftarrow)$

Since $(mes, v, t') \in s.history[w.p]$, a message must have been received with (v, t') as the clock entry for the process $P_{w.p}$. ¿From properties of the FTVC, this implies that there exists a state u in $P_{w.p}$ with that vector clock which causally precedes s. That is,

$$\exists u : u.p = w.p \wedge u.ver = w.ver \wedge u.clock[u.p] = (v,t) \wedge u \rightarrow s$$

Since $w.clock[w.p] = (v,t) \wedge (t < t')$, this implies that

$$\exists u : restored(w) \wedge u.p = w.p \wedge u.ver = w.ver \wedge w \rightarrow u \wedge u \rightarrow s$$

¿From the definition of $lost(u)$, this is equivalent to $\exists u : lost(u) \wedge u \rightarrow s$. Thus, $orphan(s)$ is true.

$(\Rightarrow)$

¿From definition, $orphan(s) \equiv \exists y : lost(y) \wedge y \rightarrow s$. Among all such y's, let u be a maximum state for a given version of a given process. Thus, there exists u such that $lost(u) \wedge u \rightarrow s$, and $\forall x : (x.p = u.p \wedge x.ver = u.ver \wedge x \neq u \wedge lost(x) \wedge x \rightarrow s) \Rightarrow x \rightarrow u \ldots (1)$

Let $u.clock[u.p] = (v,t)$. On any path from u to s, $u.p$th entry (v,t) of FTVC could not have been overwritten. From (1), it could not be overwritten by an entry from version v. For a higher version v', overwriting process would have waited for token about version v and then that process would have rolled back. This implies that $(mes, v, t) \in s.history[u.p]$. Further, $lost(u)$ implies,

$$\exists w : restored(w) \wedge w \rightarrow u \wedge w.version = u.version$$

Therefore, $\exists w : restored(w) \wedge w.clock = (v,t') \wedge (t' < t)$ ■

The next lemma gives a sufficient condition to detect an obsolete message. It also states the circumstances in which this condition is necessary.

Lemma 4 *For any message m received in state s, if there exists an entry $(token, v, t)$ in history of s for process P_j and $m.clock[j] = (v,t')$ such that $(t < t')$, then m is obsolete. That is,* $[(token, v, t) \in s.history[j] \wedge m.clock[j] = (v,t') \wedge t < t'] \Rightarrow obsolete(m)$.
This condition is also necessary when there are no undelivered tokens.

Proof: Since $(token, v, t) \in s.history[j]$, $\exists w : w.p = j \wedge restored(w) \wedge w.clock[j] = (v,t)$. ¿From FTVC algorithm and $m.clock[j] = (v,t')$, we get that $\exists u \in P_j : u.clock[j] = (v,t')$. Since $(t < t')$ and a token (v,t) exists for P_j, it follows that u is a lost state.

Let x be the state from which the message m is sent, that is $x = m.sender$. From $u.clock[j] = (v,t')$, $u \rightarrow x \vee u = x$. This implies that $lost(x) \vee orphan(x)$. That is, $obsolete(m)$

For **converse**, the definition of $obsolete(m)$ imply $lost(m.sender) \vee orphan(m.sender)$. This implies that $\exists u : restored(u) \wedge u \rightarrow m.sender$. Let $u.clock[u.p] = (v,t)$. $(u \rightarrow m.sender) \Rightarrow \{(m.sender).clock[u.p] = (v,t') \wedge (t' > t)\}$. This is because on the path from u to $m.sender$, (v,t') could not have been overwritten by an entry from higher version v' of $P_{u.p}$. Before overwriting, a process would have waited for token about $P_{u.p,v}$ and then it would have rolled back. Since all tokens have been delivered, so trivially, $\exists s : (token, v, t) \in s.history[w.p]$. ■

The above test is optimal in the sense that except for the conditions stated, a process $P_{s.p}$ will not be able to detect an obsolete message. It will accept it and as per the next lemma will become an orphan.

Lemma 5 *If a message m is obsolete and s accepts m then s is an orphan state.*

Proof:

The message m is obsolete implies that $lost(m.sender) \vee orphan(m.sender)$. That is, either $lost(m.sender)$ or there exists u such that $lost(u) \wedge u \rightarrow m.sender$. From, $m.sender \rightarrow s$, it follows that $orphan(s)$ ■

The next theorem shows that our protocol is correct.

Theorem 2 *This protocol correctly implements recovery, that is, either a process discards an obsolete message or the receiver of an obsolete message eventually rolls back to a non-orphan state.*

Proof: Let a failure of the version v of P_i cause a message m to become obsolete. If the receiver P_j has received a token about $P_{i,v}$ before receiving m, then by lemma 4, it will recognize that m is obsolete and will discard m. Otherwise, it will accept m and by lemma 5, will become an orphan. But P_j will eventually receive tha token about $P_{i,v}$. Then by lemma 3, it will recognize that it is orphan and will rollback to a non-orphan state. ■

6.8 Properties of the protocol

Theorem 3 *This protocol has following properties: asynchronous recovery, minimal rollback, handling concurrent failures, toleration of network partitioning, recovering maximum recoverable state.*

Proof:

Asynchronous Recovery: After a failure, a process restores itself and starts computing. It broadcasts a token about its failure but it does not require any response.

Minimal Rollback: In response to the failure of a given version of a given process, other processes rollback at most once. This rollback occurs on receiving the corresponding token.

Handling Concurrent Failures: In response to multiple failures, a process rolls back in the order in which it receives information about different failures. Concurrent failures have the same effect as that of multiple non-concurrent failures.

Tolerance of Network Partitioning: Because of asynchrony, a process recovers immediately from a failure. Since we assume reliable token delivery, other processes will also eventually know about this failure and will rollback, if required.

Recovering Maximum Recoverable State: Only orphan states are rolled back. ■

6.9 Overhead Analysis

Except application messages, the protocol causes no extra messages to be sent during failure-free run. The following overheads are involved in this protocol:

1. FTVC: The protocol tags a FTVC to every application message. The FTVC might be needed for purposes other than recovery, for example predicate detection [9]. Let the maximum number of failures of any process be f. The protocol adds $\log f$ bits to each timestamp in vector clock. Since we expect the number of failures to be small, $\log f$ should be small.
2. Token broadcast: A token is broadcast only when a process fails. The size of a token is equal to just one entry of vector clock. So broadcasting overhead is low.
3. History: Let the number of processes in the system be n. There are at most f versions of a process and there is one entry for each version of a process in the history. So the size of the history is $O(nf)$. The history is maintained in relatively inexpensive main memory and f is expected to be small.

7 Conclusion

Smith et. al. [25] presented the first completely asynchronous, optimistic recovery protocol. The main limitation of their work is the number of timestamps in their vector clock. We improved on their work by moving the needed information from the vector clock to volatile memory. Still, the size of FTVC may become a bottleneck in systems with thousands of processes. So a future research direction is to move even more information from FTVC to volatile memory and send only one timestamp with each message, while maintaining the asynchronous nature of optimistic recovery.

Acknowledgement

We would like to thank Tarun Anand for helping in the preparation of an earlier version of this report. We would also like to thank Craig Chase, J. Roger Mitchell, Venkat Murty, and Chakrat Skawratonand for their comments on an earlier draft of this report.

References

[1] L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and Orphan-Free Message Logging Protocols. *Proc. 23rd Fault-Tolerant Computing Symp.* , 145-154, 1993.

[2] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, and Causal. *Proc. 15th Intl. Conf. on Distributed Computing Systems*, 229-236, 1995.

[3] A. Borg, J. Baumbach, and S. Glazer. A Message System Supporting Fault Tolerance. *Proc. 9th ACM Symp. on Operating Systems Principles*, 90-99, 1983.

[4] F. Cristian and F. Jahanian. A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations. *Proc. 10th IEEE Symp. on Reliable Distributed Systems*, 12-20, 1991.

[5] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, 3(1): 63-75, Feb. 1985.

[6] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent Rollback Recovery with Low Overhead, Limited Rollback and Fast Output Commit. *IEEE Trans. on Computers*, 41 (5): 526-531, May 1992.

[7] L. Gunaseelan and R. J. LeBlanc, Jr. Event Ordering in a Shared Memory Distributed System. *Proc. 13th Intl. Conf. on Distributed Computing Systems*, 256-263, 1993.

[8] V. K. Garg and A. I. Tomlinson. Using Induction to Prove Properties of distributed Programs. *Proc. 5th IEEE Symp. on Parallel and Distributed Processing*, 478-485, 1993.

[9] V. K. Garg and B. Waldecker. Detection of Weak Unstable Predicates in Distributed Programs. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 3, pp. 299-307, March 1994.

[10] D. B. Johnson. Efficient Transparent Optimistic Rollback Recovery for Distributed Application Programs. *Proc. 12th IEEE Symp. on Reliable Distributed Systems*, 86-95, 1993.

[11] D. B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. *Proc. 17th Intl. Symp. on Fault-Tolerant Computing*, 14-19, 1987.

[12] D.B. Johnson and W. Zwaenpeoel. Recovery in Distributed Systems using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11: 462-491, September 1990.

[13] R. Koo and S. Toueg. Checkpointing and Rollback Recovery for Distributed Systems. *IEEE Trans. on Software Engineering*, 13(1): 23-31, Jan. 1987.

[14] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, vol. 21, no. 7, 558-565, 1978.

[15] B. W. Lampson. Atomic Transactions. *B. W. Lampson, M. Paul, and H. J. Siegert, editors, Distributed Systems-Architecture and Implementation, Spring-Verlag*, 246-265, 1981.

[16] H. V. Leong and D. Agrawal. Using Message Semantics to Reduce Rollback in Optimistic Message Logging Recovery Schemes. *Proc. 14th Intl. Conf. on Distributed Computing Systems*, 227-234, 1994.

[17] F. Mattern. Virtual Time and Global States of Distributed Systems. *Parallel and Distributed Algorithms: Proc. of the Intl. Workshop on Parallel and Distributed Algorithms, Elsevier Science Publishers B. V.(North Holland)*, 215-226, 1989.

[18] N. Neves, M. Castro, and P. Guedes. A Checkpoint Protocol for an Entry Consistent Shared Memory System. *Proc. 13th ACM Symp. on Principles of Distributed Computing*, 121-129, 1994.

[19] S.L. Peterson and P. Kearns. Rollback Based on Vector Time. *Proc. 12th IEEE Symp. on Reliable Distributed Systems*, 68-77, 1993.

[20] M. L. Powell and D. L. Presotto. Publishing: A Reliable Broadcast Communication Mechanism. *Proc. 9th ACM Symp. on Operating Systems Principles*, 100-109, 1983.

[21] B. Randell. System Structure for Software Fault Tolerance. *IEEE Trans. on Software Engineering*, 1(2): 220-232, June 1975.

[22] D. L. Russel. State Restoration in Systems of Communicating Processes. *IEEE Trans. on Software Engineering*, 6(2): 183-194, March 1980.

[23] G. G. Richard III and M. Singhal. Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory. *Proc. 12th IEEE Symp. on Reliable Distributed Systems*, 58-67, 1993.

[24] G. Suri, B. Janssens, and W. K. Fuchs. Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory. *Proc. 25th Intl. Symp. on Fault-Tolerant Computing*, 279-288, 1995.

[25] S. W. Smith, D. B. Johnson, and J. D. Tygar. Completely Asynchronous Optimistic Recovery with Minimal Rollbacks. *Proc. 25th Intl. Symp. on Fault-Tolerant Computing*, 361-370, 1995.

[26] A. P. Sistla and J. L. Welch. Efficient Distributed Recovery Using Message Logging. *Proc. 8th ACM Symp. on Principles of Distributed Computing*, 223-238, 1989.

[27] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. on Computer Systems*, 204-226, August 1985.

[28] Y. M. Wang, P. Y. Chung, I. J. Lin, and W. K. Fuchs. Checkpointing Space Reclamation for Uncoordinated Checkpointing in Message Passing Systems. *IEEE Trans. on Parallel and distributed Systems*, vol. 6, no. 5, 546-554, May 1995.