Group message logging without orphans for fast output commit

J. Roger Mitchell and Vijay K. Garg

TR-PDS-1997-003 **March 1997**

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

Group message logging without orphans for fast output commit

J. Roger Mitchell[*] Vijay K. Garg[†]

Parallel and Distributed Systems Laboratory
email: pdslab@ece.utexas.edu
Department of Electrical and Computer Engineering
The University of Texas at Austin,
Austin, TX 78712-1084

March 1997

Abstract

In current recovery algorithms, the problem of output commit is often not specifically addressed. Those schemes that do, incur high latency before the output message can be sent from the system. This latency typically involves synchronous writes to stable storage. In this paper we present a method of grouping processes which allows output commit to occur after a process has received acknowledgements from other processes. We keep the number of acknowledgements small by creating groups of processes within the system, each group i tolerates k_i simultaneous failures. Our experiments show that when $k_i = 2$ or 3, our algorithm performs output commit over three times faster than simply writing information to stable storage. Our algorithm is based on the family-based message logging approach, but our approach allows for k_i simultaneous failures in each group, rather than just k across the entire system. Our implementation of family-based logging also never blocks live processes and allows for independent recovery during simultaneous failures, something that previous work has not accomplished. We introduce the idea of a recovery reply cut to do this.

1 Introduction

In a distributed system, shared or foreign-owned resources present a problem for recovery schemes. Since these resources include servers which are 'owned' by some other entity (for which it is not possible to impose a recovery scheme) a recoverable computation must communicate with these resources as it would the "outside world." That is, messages sent to these resources are irretrievable. This is the problem of *output commit*. Most current recovery algorithms do not focus on this problem, and those that do, incur high latency before the send can take place. The fact that shared resources may use their own fault tolerance scheme is another reason low latency output

[*]supported in part by a Virginia & Ernest Cockrell fellowship

[†]supported in part by the NSF Grants ECS-9414780, CCR-9520540, a TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant

commit is desireable. This scenario will become more prevalent as distributed objects become more widespread [12]. The recovery algorithm presented here is an attempt to decrease the latency of output commit.

Current recovery schemes handle output commit differently. In consistent checkpointing, all processes take a checkpoint together. There is coordination so that the set of checkpoints stored comprises a consistent state [6, 13]. To perform output commit, this coordinated save of checkpoints must occur; a very long delay.

Message logging recovery schemes log the order in which messages were received. By assuming receive ordering is the only non-determinism, execution is recoverable using this ordering. No coordination of checkpointing is required in these schemes. Pessimistic message logging [5, 18] forces a process to wait before sending any message while the message log is written to stable storage. In pessimistic logging, an output commit send is like any other send, it must wait for the log to be written. Optimistic logging methods [8, 15, 19, 20, 22] 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 which depend on them may not be recoverable. This may then require that unfailed processes roll back their execution as well. Because of this optimistic operation, before performing output commit, processes with execution which may have caused the send must save their state to stable storage. 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 [9] uses this method. In family based message logging (FBL) [3] 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.) For both of the causal logging approaches, before performing output commit, the sender must write the causal history of the send to stable storage before sending the message. Finally, there is a message logging approach which sends the ordering information around a ring [7], which we call *ring logging*. In this case, the commit does not require a write to stable storage, but does require a delay of $2n$ message hops and $2n$ additional messages, where n is the number of processes participating in the recovery scheme.

We believe, like optimistic approaches, that failures are rare, and like Alvisi and Marzullo in [2, 3], that optimizing for small numbers of simultaneous failures and avoiding rolling back unfailed processes best matches real-world needs. Therefore, we present a recovery method based on the FBL approach. In our approach, however, output commit can be performed using send and acknowledge messages, without writing to stable storage, and the system is divided into groups. We give experimental data that shows that for 2 or 3 simultaneous failures per group, send/ack can be done in less than one third the time it takes to write to stable storage. The number of allowed simultaneous failures in group i is k_i, while the total in the system is $\Sigma_{i \in G} k_i$, where G is the set of groups. We have also developed a recovery method for FBL which avoids blocking unfailed processes or requiring that simultaneous failures recover together. The focus of the research is on systems in which output commit occurs often.

In the next section we compare our work with the other receive order logging methods and with previous efforts in recovery with output commit. In section 2 we present the two necessary properties for a recovery algorithm to be consistent and allow output commit. Then we present the algorithm, its correctness, and give experimental results.

1.1 Comparison to previous work

As just stated, our scheme is similar to work done by Chiu and Young [7] and Alvisi and Marzullo [2, 3]. Chui and Young present the ring method of passing receive ordering information around a ring of processes. This method, however, requires that unfailed processes roll back during certain failures. Their scheme also blocks certain message receives at a process when that process attempts to take a checkpoint. Our scheme has neither of these shortcomings. They also assume reliable channels, which we do not. Finally, as mentioned before, their approach sends a message around the ring of N processes twice to commit and once for every message sent. Alvisi and Marzullo present family-based message logging which does not require unfailed processes to roll back. Our scheme is based on FBL. In our approach, however, we break the system into groups and perform the FBL scheme within each group. FBL scatters logging information across the system, whereas our method maintains the information close to the originator and in fewer processes. When performing output commit, FBL must block while stable storage is written. We show that sending the log within the group can be faster. Their approach can handle K failures, while our approach actually tolerates $\Sigma_{l \in L} K_l$ failures. In addition, our approach allows a system designer to choose which processes will have greater fault tolerance, something which is not possible in other approaches. FBL in [2, 3] requires that live processes block during recovery, while in [11], Elnozahy requires simultaneous failures must coordinate to recover. Our implementation never blocks live processes and simultaneous failures recover individually. Table 1 compares our recovery scheme with the other causal message logging schemes. f is the number of serial failures which occur during another recovery ($f < K$).

Research	Latency of output commit	Latencies caused by recovery	Configurability
Alvisi, et.al. [1]	Write to stable storage	Unfailed procs must block	Choice of level of failures
Elnozahy [11]	Write to stable storage	Recovery must detect other failures, agree as to failure order, perform f phase message exhange	Not addressed
Our approach	Exchange of messages	Recovery must perform f phase message exhange	Choice of level of failures within a group, choice of groups.

Table 1: Comparison of causal message logging schemes.

Addressing the output commit problem in process recovery schemes has been done by Elnozahy and Zwaenepoel [9] and Johnson [16]. The first paper uses antecedence graphs to maintain message ordering information. In this scheme, every message sent includes an antecedence graph. To send output commit, the algorithm must block until the antecedence graph at the process is written to stable storage. Johnson's approach for output commit [16] can be used with optimistic logging or consistent checkpointing. Johnson does not add as much information to application messages as Elnozahy and Zwaenepoel. However, to commit, the scheme must block while all processes with states which are in the causal history of the send ensure their states are on stable storage.

We give a graph at the end of the chapter that demonstrates that our approach to output commit can be faster than any approach that requires a write to stable storage.

Creating groups of processes within the system for recovery was also mentioned by Sistla and Welch in [19]. They use groups with optimistic message logging to avoid nodes far removed from the failed node from having to roll back. We use groups with causal message logging, to decrease latency of output commit and time of recovery.

2 Model

Our algorithm operates in an environment in which processes may fail by crashing. They may be restored by *rolling back* to a previous state. Every time a process fails and rolls back it increments its *incarnation* number. Like other message logging approaches, we assume deterministic execution. Messages may be unreliable and channels are not ordered. However, if the number of simultaneous message and process failures is greater than k within a group, a message for the outside may not get sent. P_i and $\mathcal{P}$ represent process i and the set of N processes running the algorithm respectively. For ordering of states where processes can recover, we use a modified precedes relation similar to [8] as follows:

Definition 2.1 *(***Precedes***)*
A state s precedes a state t, denoted by $s \prec t$, iff:

1. *s and t are on the same process and s occurred before t.*
2. *the following is not true: s is in an earlier incarnation than t, and s has been rolled back.*

We also define the happened before relation on states similar to Lamport's happened before relation on events. We do this for our environment where processes recover.

Definition 2.2 *(***Happened before***) A state s happened before a state t, denoted by $s \rightarrow t$, is the smallest relation satisfying one of the following:*

1. *$s \prec t$*
2. *s is the state immediately preceding the send of a message and t is the state immediately following the receive of that message.*
3. *there exists a state u such that $s \rightarrow u$ and $u \rightarrow t$.*

A process P_i performs a sequence of events, e_i's, and passes through a sequence of states, s_i's: $s_{i,1}, e_{i,1}, s_{i,2}, e_{i,2}, ...e_{i,l-1}, s_{i,l}$. We call this $execution_i$. We describe an abstraction run_i which is made up of only those states and events in $execution_i$ which are never rolled back. Therefore, only in run_i are all events ordered by causality since rolled back events do not causally precede any event that was not rolled back. We define $execution_{\mathcal{P}} = \cup_{i \in \mathcal{P}}\ execution_i$. We also define $run_{\mathcal{P}} = \cup_{i \in \mathcal{P}}\ run_i$.

3 Properties for recovery with output commit

Note that in any $run_{\mathcal{P}}$, if a receive event is in the run, the corresponding send event must also be in the run. Let E be the set of events, then this property can be stated as:

- **Consistency:** All events which causally precede non rolled back events are also not rolled back.

$$\forall e, f \in E : f \in run_{\mathcal{P}} \wedge e \rightarrow f \Rightarrow e \in run_{\mathcal{P}}$$

This is particularly relevant in recovery protocols. Some recovery methods create *orphan* events when a failure occurs. An orphan event is an event which causally follows a send for which the send is no longer part of the run (due to a rollback). For example, if P_j were to roll back past a send event e, then $e \notin run_j$. A recovery protocol must enforce the consistency property so that it meets the requirement of "no orphan events in run".

To perform output commit, the commit event (or send) must not be removed from $run_{\mathcal{P}}$. Therefore, at P_i, all commit events must be a part of run_i. Note that any recoverable event of P_i is an event which is part of run_i. We state the property as:

- **Safety:** A process will not perform output commit unless it is guaranteed to be recoverable:

$$\forall e \in E, i : (e \in execution_i) \wedge (e = \text{commit event}) \Rightarrow e \in run_i$$

4 Group message logging

For our algorithm, all processes are placed in groups, where a group l can tolerate K_l failures. This is done for several reasons:

1. To enable K_l to be smaller than K required for the entire system, so that output commit has smaller latency, and messages within a group have less overhead. (Recall that $\Sigma_{l \in L} K_l$ total failures are allowed, so a smaller group can have smaller K_l.)
2. To keep logging information close to the process logged and help localize recovery.

Groups have the following properties:

- The size of group l can vary from $2K_l+1$ to N. A lower bound of $2K_l+1$ allows our algorithm to be non-blocking and still tolerate K_l failures per group.
- Except for message resends, recovery in one group is independent of recovery in another.

- When $K_l = 1$ for group l, delivery of a message outside of l is sent via another process in the group. When $K_l > 1$, delivery of a message outside group l (including output commit) requires the send of information to other processes in the group and the receive of K_l acknowledgements.

To decrease latency for intergroup messages, processes that communicate often can be put in the same group, and smaller groups can be created with small K_l. Within a group, we employ message logging similar to FBL. So, when one group is used for the entire system, the algorithm reduces to FBL causal message logging with special procedures for output commit and recovery. Since K_l can vary for different l, critical processes can have higher fault tolerance. Non-critical processes can be put outside of all groups.

We next give an example of the algorithm operation and then present the algorithm formally. We use K rather than K_l in the remainder of the paper.

4.1 Considerations for groups

To group processes, there are considerations which should be made to best take advantage of what group message logging can offer.

- Processes which will be communicating often should be placed in the same group. Examples are a set of processes performing a distributed task or processes performing coordination.

- Processes which are on the same host need to be considered carefully. If, for example, a set of processes are on the same host and communicate often (which will often be the case), it is desireable to have them in the same group. However, this group should include processes on other hosts and K should be set at least as large as the maximum number of processes which are on the same host in this group. This allows recovery to occur when the host fails.

- K is the maximum number of simultaneous failures that are expected to occur in a group. Because the computation may be spread over a large number of hosts, there may be different factors which affect this, such as exposure to environmental affects or simply robustness of software. With different groups, different values of K may be chosen for each based on these factors.

4.2 Example and operation of algorithm

Here we will present an example of our algorithm's operation. We present the general case for K but note that when $K = 1$ the algorithm has optimizations which reduce messages and latency (see appendix).

The operation of a group of processes in our method is simple: a message is not sent from the group until $K+1$ processes have the message receive ordering information necessary to recover the sender. As with FBL causal message logging, a message sent within one of our groups includes any receive ordering information held at the sender that is not already held by $K+1$ processes. By doing this, if K failures occur within a group, these K processes can recover. To implement the causal message logging scheme, each process occasionally takes a checkpoint of its current state and saves it to stable storage. A process does not remove a checkpoint from stable storage until the next has been written. The message receive ordering since the last checkpoint is maintained in a *receive list* in volatile memory. Similar to [2, 3] the entry for a message in this list is called a *determinant*. The determinant contains information such as sender, message number, and receive order. In [4] Alvisi discussed different methods for determining when it was necessary to send a determinant. These approaches were based on FIFO channels, which we do not assume, however, many of the ideas are applicable here. Here we use a simple approach of keeping track of the number of processes known to hold the determinant.

To send a message within a group, as in family based logging [3], the message includes all determinants which are not already at $K+1$ processes. To send a message outside the group, a process first sends determinants to $2K$ other processes in the group and waits for K acknowledgements before sending. This ensures that these determinants are at $K+1$ processes in the group, and that failures never block the send.

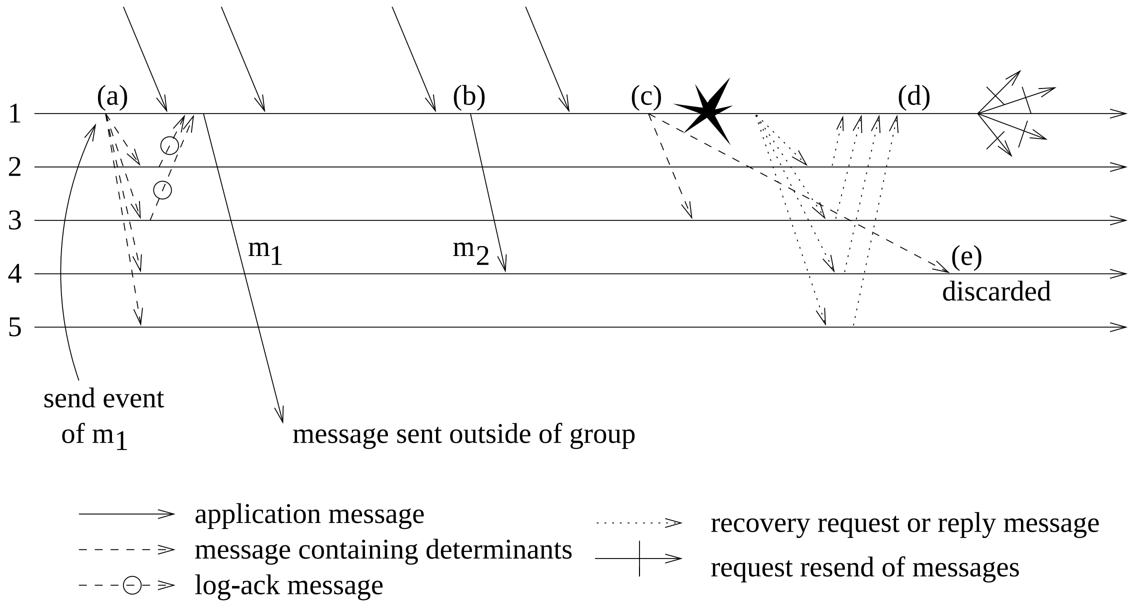

Figure 1: The send and recovery operation for a group with $k = 2$.

Figure 1 shows the operation of a group for which $K = 2$. At (a), P_1 desires to send m_1 outside the group. Before doing so, any necessary determinants are sent to $2K$ $(= 4)$ other processes. When K $(= 2)$ acknowledgements have been received, m_1 is sent. At (b), P_1 sends a message within the group. This message contains determinants not yet at $K+1$ processes. At (c) P_1 again attempts to send a message outside the group. However, it fails before receiving k acknowledgements and recovers at (d). This recovery includes collecting determinants from the group in recovery reply messages and requesting duplicates of application messages based on the determinants received. The event (e) shows what happens when events cross failure boundaries. Later in the paper we will refer to this boundary as a recovery reply cut. Because the message at (c) was sent before P_1's failure and P_4 has already responded to this failure, the message is discarded. This avoids orphan events.

4.3 Algorithm

Our approach tolerates K simultaneous failures within a group, where the size of the group can be $2K+1$ to N. In this section a general algorithm for K is given. We give optimizations for $K=1$ in the appendix. The optimizations do not require the same send/acknowledge approach for output commit. For all K, a process is said to be *recovering* if it has failed and has not yet received *valid* recovery reply messages from all processes in the group. A valid recovery reply message is one that has up to date information on failures (see condition R3 of the algorithm). Figure 2 shows this graphically. We can now give the limit to which our algorithm operates correctly in terms of

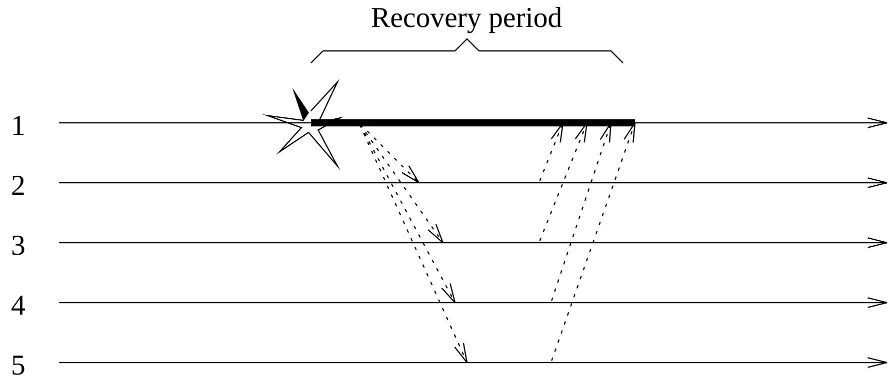

Figure 2: A process that is recovering

multiple failures within a single group.

> A process that is *recovering* can recover correctly if no more than $K-1$ recovery reply messages it receives were sent from processes that were also *recovering*.

If this limit is exceeded then greater than K processes are simultaneously *recovering*. Although the algorithm will continue to satisfy the properties of safety and consistency, no guarantees can be made if this condition occurs. Since it is assumed that failed processes will eventually enter recovery, any recovering process will eventually receive all reply messages. Note that failed processes do not block normal execution.

4.3.1 Maintaining receive order

To specify our algorithm we now define the following for a message m:

n_m	the message number of m.
s_m	the sender of m.
d_m	the destination of m.
r_m	the receive number of m.
h_m	the minimum number of processes holding the determinant of m.
D_m	the determinant of m which is a tuple, $(n_m, s_m, d_m, r_m, h_m)$. These entries for any determinant, D, are accessed as $D.n, D.s, D.d, D.r, D.h$.
num_acks$_m$	the number of *log-ack* messages received for m.

```
upon message m to send to P_j do                                                SR1
    msgs_sent_k = msgs_sent_k + 1;
    for all (D ∈ 𝒟_k : D.h ≤ K)
        add D 𝒟_m;
    if (P_j within group) then
        send (f_k[k], 𝒟_m, msgs_sent_i) with m as application message to P_j;
    else
        send (f_k[k], 𝒟_m, 0) as group message to 2K processes within group;
    save m, d_m = P_j, and n_m = msgs_sent_k for sending/resending;
upon receiving group message from P_j and recovering=FALSE do                   SR2
    remove (f_m, 𝒟_m, n_m) from m;
    if (f_m = f_k[j]) then
        send log-ack_m to P_j for m;
        get_determinants(𝒟_m);
upon receiving log-ack_m and m has not been sent do                             SR3
    num_acks_m = num_acks_m + 1;
    if ((num_acks_m = K) then                                                   SR4
        send (f_k[k], ∅, n_m) with m to d_m as application message;
        for all (D ∈ 𝒟_m)D.h = K + 1;
upon receiving application message, m, from P_j and recovering=FALSE do         SR5
    remove (f_m, 𝒟_m, n_m) from m;
    if ((P_j within group and f_m = f_k[j]) or (P_j not in group) then
        msgs_rcvd_k = msgs_rcvd_k + 1;
        get_determinants(𝒟_m);
        deliver m;
        𝒟_k = 𝒟_k ∪ (n_m, P_j, P_k, msgs_rcvd_k, 1); /* create new determinant. */
```

Figure 3: Send/receive rules for P_k.

We also define:

msgs_sent$_k$	the number of messages sent by P_k.
msgs_rcvd$_k$	the number of messages received by P_k.
num_replies$_k$	the number of recovery replies received for the latest failure at P_k.
f_k	the incarnation vector at P_k. $f_k[j]$ is the number of failures of P_j known by P_k (initially 0). The size of f_k equals the size of P_k's group.
$\mathcal{D}_k$	the set of determinants held by P_k.
group_size$_k$	the size of P_k's group.

The function get_determinants($\mathcal{D}$) is given in figure 4. This function does the following: for all $D \in \mathcal{D}$, if $D \notin \mathcal{D}_i$, add D to $\mathcal{D}_i$, otherwise for the $D' = D$ set $D'.h$ to the maximum of $D.h$ and $D'.h$.

```
get_determinants(𝒟)
    for all D ∈ 𝒟 do
        if (∄D′ ∈ 𝒟_k : D′.s = D.s ∧ D′.n = D.n) then
            D.h = D.h + 1;
            𝒟_k = 𝒟_k ∪ D;
        else D′.h = maximum of D′.h and D.h;
```

Figure 4: Function get_determinants at P_k.

The send/receive portion of the algorithm is given in figure 3. When sending a message m, we include ($f_k[k]$, any necessary determinants, n_m). A description follows:

- **SR1**: *sending an application message* – If the destination of the message is within the group then send the incarnation and all determinants having $h_m < K$ with the message to the destination. Otherwise, send a group message with these same determinants to $2K$ other processes in the group.
- **SR2**: *receiving a group message* – If the incarnation number of the message is current then send log-ack to the sender and remove and save determinants. Otherwise, discard message.
- **SR3**: *receiving log-ack message for m* – Increment the acknowledgements received for m.
- **SR4**: *sending message outside of group* – If the number of log-acks received for m is equal to k, then send m to destination.
- **SR5**: *receiving application message* – If the message is from within the group and the incarnation number is current then deliver the message and for all determinants received, increment h_m and save them. If the message is from another group, deliver message. In both cases create a determinant for the receive, and save it. Otherwise, discard message.

4.3.2 Recovering after failures

When a process fails in causal message logging algorithms, to recover it must recreate its message log (that is, the receive ordering) and at the same time ensure any future receives do not become orphans. This second point requires consideration.

First the scheme must ensure that upon recovery messages from the recovering process do not create orphan events. Figure 5 illustrates how this might happen. In the figure, P_1, upon restart, requests all determinants. P_2 replies with the determinant set it has for P_1. 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 goes on to receive m' it will create an orphan event because the receive of m has not been recreated by P_1.

The solution to the problem is for P_1 to include, in all messages sent, the number of the incarnation from which any message is sent. This is done in our algorithm of figure 3. When recovery information is requested, P_1 indicates that it is entering a new incarnation, and any messages subsequently received from earlier incarnations should be discarded. In the figure, P_2 would know to discard m'.

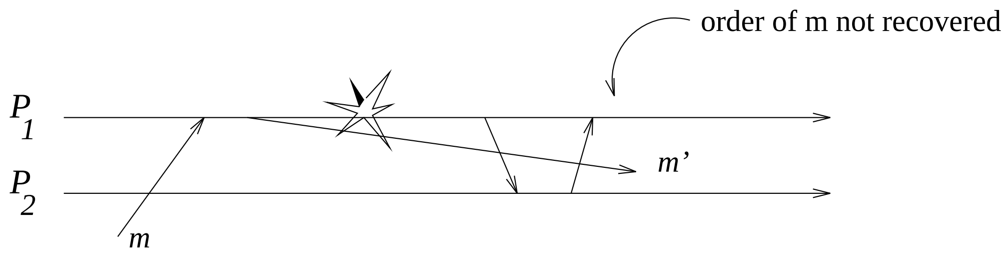

Figure 5: Possible orphan creation

The problem extends itself to cases when there are multiple failures. Figure 6 demonstrates this. Here, even though P_1 has indicated its new incarnation number, P_3 can still create an orphan event by receiving m''. This is because it does not yet know about P_2's failure.

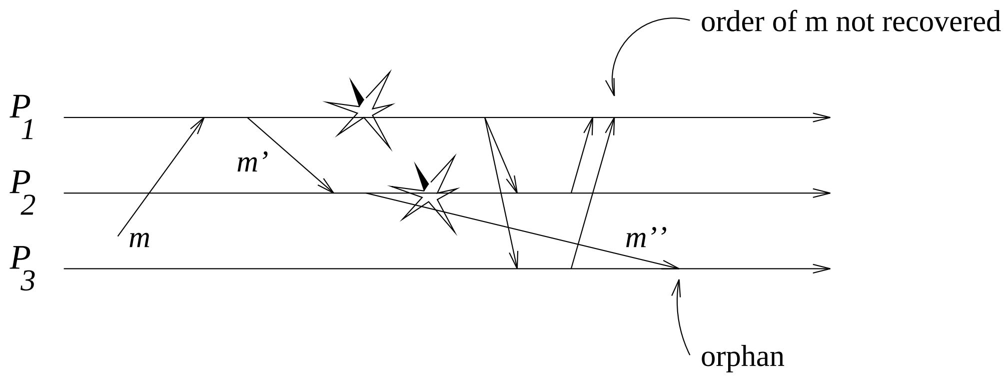

Figure 6: The problem with simultaneous failures

One solution to this problem is to have any process which responds with a recovery reply, block temporarily while it writes its reply to stable storage. This process can then be sure that it always knows the determinant set included in the reply. The process then blocks any receives from being delivered until it receives a message from the process to which it sent the reply indicating recovery is complete [4, 15]. Within this message are listed all messages for which determinants were received and that are now part of the recovered process's run. Figure 7 shows this scenario. 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.

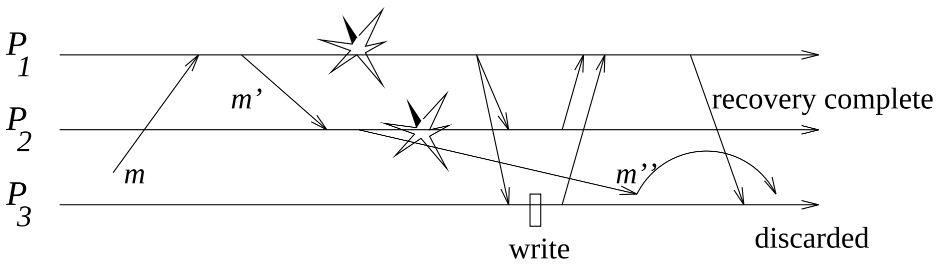

Figure 7: One solution to the problem of simultaneous failures

Another solution was presented by Elnozahy [11]. Here a process upon recovery reads and increments its incarnation number from/to stable storage. It then assigns itself a number, *ord*, which means it was the *ord*th process failure in the system. From the set of currently recovering processes, the process with the lowest *ord* becomes the leader. It 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 it to discard messages that originated from earlier incarnations. The live processes respond to the incarnation messages by sending all determinant information to the leader. The leader then forwards this to all recovering processes. This is shown in figure 8. 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 9 shows this.

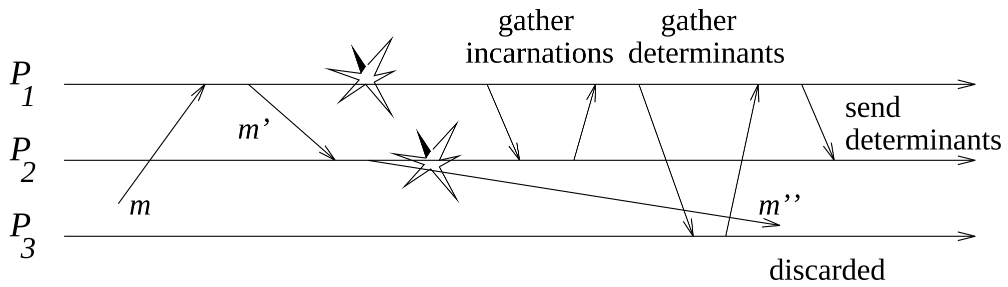

Figure 8: Elnozahy's solution

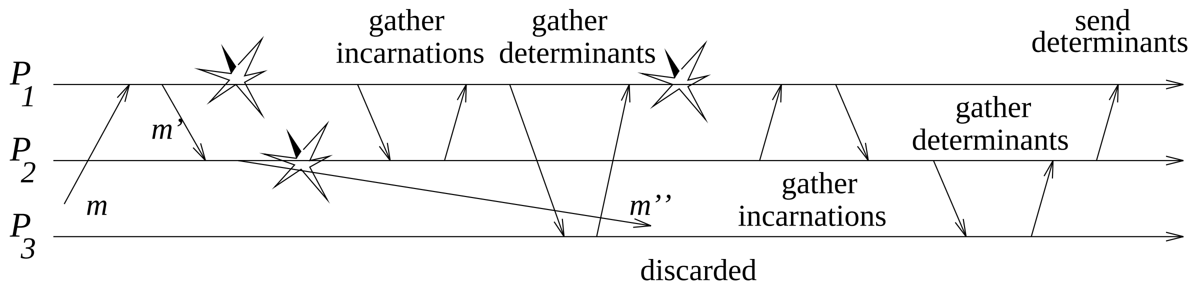

Figure 9: Multiple failures can cause all recoveries to block

There are several problems with this approach:

- Requires knowledge of recovering processes. This knowledge would have to be manifested as a broadcast message from a process which begins to recover. The broadcast would indicate that it is part of the recovering set.
- Requires implied agreement as to order of failures.

Since recovery detection and ordering are not free, messages must be exchanged to perform these tasks in a message passing environment in Elnozahy's method. Suppose a process upon recovering performs recovery detection by sending "need to recover" messages. In response, all processes acknowledge with a message which indicates whether it is recovering or not. If two processes fail simultaneously, they must also agree as to the order of their failure. Figure 10 shows what the recovery might actually be like to do this.

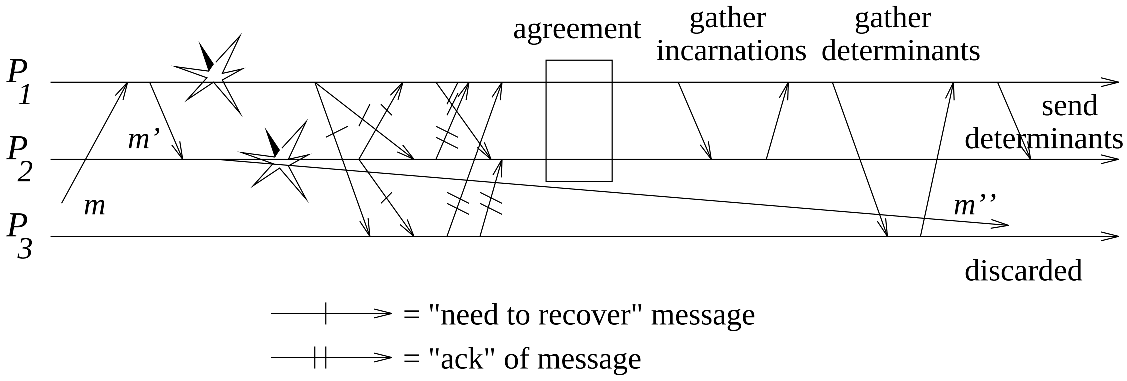

Figure 10: Recovery in which failure discovery and agreement are not implied.

For our approach, a failed process, upon recovery, reads and increments its incarnation number from/to stable storage as above. Then it sends its incarnation number to all processes. Upon receipt of the message a process updates a maintained incarnation vector. It then sends its incarnation vector with determinant information to the recovering process. If all incarnation vectors agree, the recovering process is done. However, if a reply message is received that has an incarnation vector which is less than what has already been recieved, the reply is discarded and rerequested with the current incarnation vector. If a reply is received with incarnation greater than what has already been received, the reply is accepted and a new request is sent to all earlier received replies. Figure 11 shows how the problem of m'' is solved with this approach. Figure 12 demonstrates that the repeated failure of one process does not affect the recovery of another process. Therefore, our

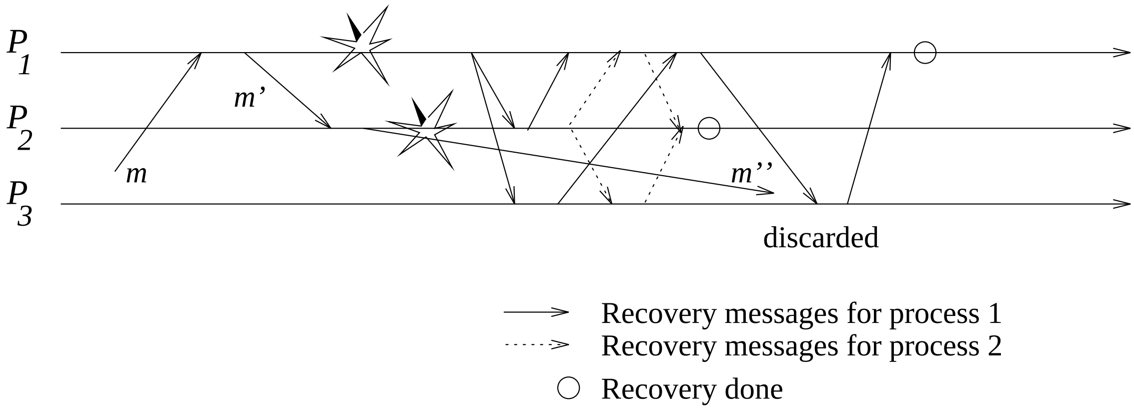

Figure 11: Our solution

approach has the following benefits:

- 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 which respond with determinant information must do so with knowledge of which incarnation the other replies are

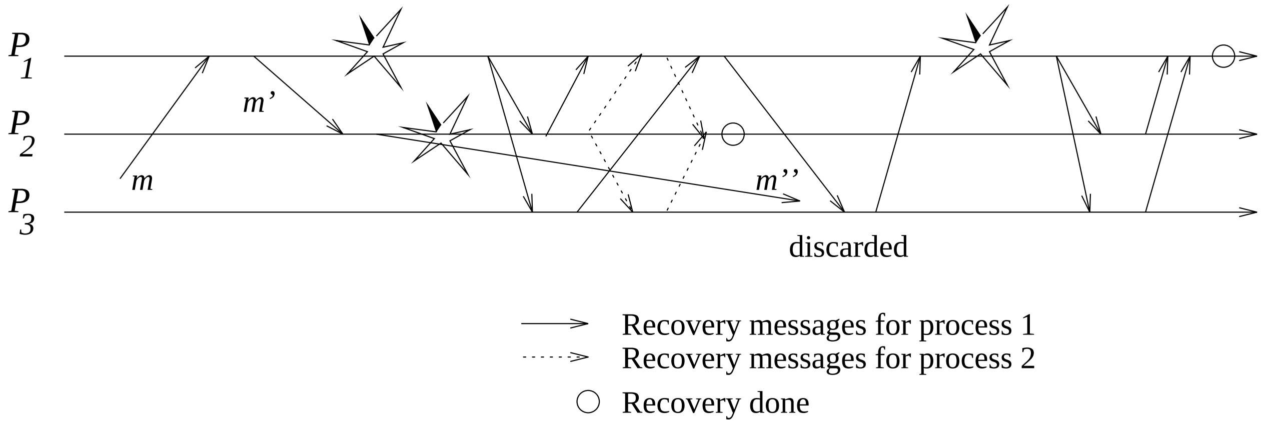

Figure 12: Multiple failures do not cause all recoveries to block

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.*

Definition 4.1 recovery reply state (RRS) for failure x **at** P_k — *The earliest global state where* $\forall i$ *in* P_k*'s group:* $f_i = f_k$*.*

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.

The recovery code for our algorithm ensures that the replies create a recovery reply state. This is given in figure 13.

- **R1**: *failure* – increment incarnation and send as recover request to all processes in group.
- **R2**: *receiving recover request* – Update incarnation and send all determinants to sender as recovery reply.
- **R3**: *receiving recovery reply* – If the received incarnation vector for any entry is less than f_k, then reject the recovery reply and request it again. If f_k for any entry is less than the received incarnation vector, then request replies again from all processes for which a reply has already been received.
- **R4**: *recovering* – Increment the number of replies received and save the determinants from message.
- **R5**: *recovering* – If a valid recovery reply has been received from every member of the group and the checkpoint is restored then request messages necessary for recovery as determined by the reconstructed message log.
- **R6**: *receiving resend request* – Resend all requested messages to recovering sender.
- **R7**: *receiving resend reply* – If all requested messages that must be delivered before this message have been, then deliver the message. If all messages required to be resent have been delivered, then the process is finished recovering.

upon failure do R1

    $recovering$ = TRUE;
    for all $l : f_k[l] = 0$;   num_replies$_k$=0;
    read $f_k[k]$ from stable storage, increment and replace;
    begin recovery of checkpoint;
    send f_k with *recover request* to all processes in group;

upon receiving *recover request* from P_j do R2

    for all l: $f_k[l]$ = max($f_k[l], f_j[l]$);
    send f_k and $\mathcal{D}_k$ to P_j as *recovery reply*;

upon receiving *recovery reply* from P_j do R3

    remove f_j and $\mathcal{D}_j$ from *recovery reply*;
    if ($\exists l : f_j[l] < f_k[l]$) then
        send f_k with recover request to P_j; /* discard recovery reply. */
    else do
        if ($\exists l : f_j[l] > f_k[l]$) then
            $f_k = f_j$;   num_replies$_k$ = 0;
            send f_k with *recover request* to all processes which have already replied;
        num_replies$_k$ = num_replies$_k + 1$; R4
        get_determinants($\mathcal{D}_j$);
        if (num_replies$_k$ = group_size$_k - 1$ and checkpoint restored) then R5
            for all $m : k = d_m$, r_m after checkpoint, and $(n_m, s_m, d_m, r_m, h_m) \in \mathcal{D}_k$ do
                if ($m \notin \mathcal{D}_k$) send n_m as part of *resend request* to s_m;

upon receiving *resend request* from P_j do R6

    for all n_m in resend request, send m to P_j as *resend reply*;

upon receiving *resend reply* do R7

    if (for all m': $r_{m'}$ after checkpoint and $r_{m'} < r_m$: m' has been delivered) then
        deliver m;
    if (for all m': $r_{m'}$ after checkpoint: m' has been delivered) then
        $recovering$ = FALSE;

Figure 13: Recovery rules for P_k.

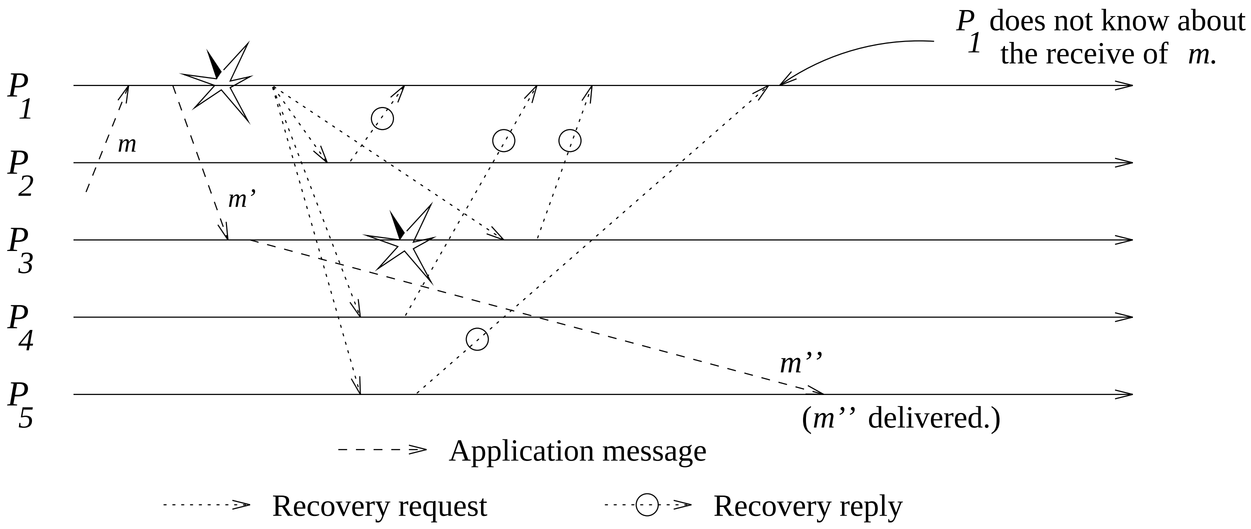

Figure 14: An inconsistency created when rule R3 not present.

$R3$ of the recovery code ensures that recovery reply messages are all received with the same view of incarnations. Again, figure 14 shows what might happen if this check were not performed.

We will show in the next section that maintaining an RRS keeps the run consistent. Figure 15 gives an example of a recovery reply state for a failure at P_1. The recovery replies received at $a, b,$

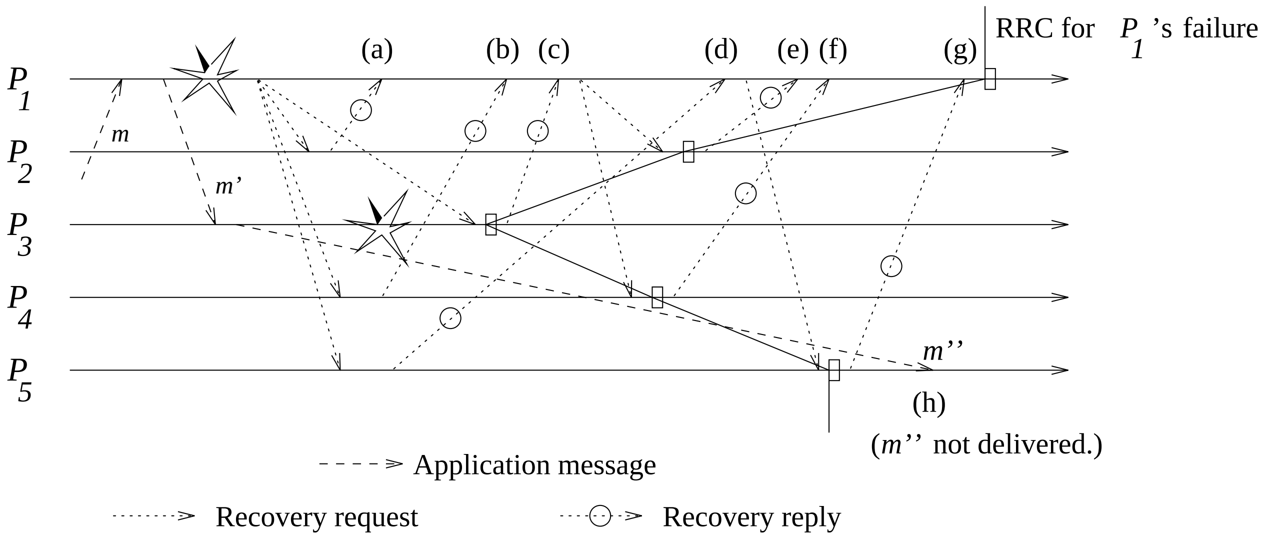

Figure 15: Example of a *recovery reply state*.

and d are invalid (and in fact the reply at d 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 c.) Therefore, P_1 updates f_i and requests recovery info from P_2 and P_4 again and from P_5 when its reply is later received at d. P_1 receives the second replies at e, f, g. P_1 also reaches R5 at g. The solid line in the figure therefore represents the recovery reply state for P_1's failure.

5 Correctness

We now show that our algorithm satisfies the properties of consistency and safety.

In our algorithm the contents of received messages are not saved, instead determinants are created. The contents of messages are saved in volatile memory at the sender. To replay execution after a failure, (and therefore regenerate messages) a process must rereceive messages specified in the recovered determinants. We demonstrate that this can be done.

Lemma 5.1 *Any message can be resent if determinants are never lost.*

Proof: It is necessary to show that no processes are ever dependent on each other for the recreation of message contents. This can be done without referring to our algorithm. Since we know that unfailed processes can resend any message, we only need to consider failed processes. Suppose a process P_1 sends a message m_1 and P_2 sends m_2. The dependency of P_2 on P_1 is true if: $m_1.r \rightarrow m_2.send$. That is, the send of m_2 is dependent on the receive of m_1.

Now, if our lemma were not true then there are D failed processes which are circularly dependent on each other. Let m_i be a message sent from P_i, such that the first is dependent on the second, etc. This can be written as:

$$m_1.receive \rightarrow m_2.send \rightarrow m_2.receive \rightarrow ... \rightarrow m_D.send \rightarrow m_D.receive \rightarrow m_1.send$$

This equation implies that $\forall i : m_i.receive \rightarrow m_i.send$, which is impossible. Therefore, $\exists k$, $\forall i \neq k$: $m_i.send \not\rightarrow m_k.send$. The contents of this m_k can be created by the checkpoint restored at P_k and messages from unfailed processes. This implies that for all i, m_i can be recreated and resent. $\Box$

The following lemma shows that for every failure a recovery reply state exists.

Lemma 5.2 *For every failure, a recovery reply state (RRS) exists.*

Proof: Every process, P_k, which fails also restarts, by assumption. Every restart sends a recovery request message to all processes in group (R1) and continues to retry until a response is received (by message suspicion and recovery scheme). Therefore, every process in the group eventually knows of the new incarnation. At most K failures can occur. Therefore, there is a finite number of changes that can occur to f_k at P_k when P_k is trying to recover. Because of these two facts, there is a state at every process P_i in the group, in which $f_i = f_k$. Therefore, RRS exists for the failure at P_k. We know that this RRS is the earliest global state in which $f_i = f_k$ at every P_i, because there is an earliest time at which $f_i = f_k$. This state is no earlier than when P_i first hears of P_k's failure. $\Box$

We will show in the next lemma that messages such as m'' in figure 15 (for which there is a failure which follows the send on the same process and precedes the recovery reply state) will not be delivered after the cut.

Lemma 5.3 *For any failure at P_i and the corresponding recovery reply state, RRS,*
$\forall m$ sent from P_j to P_k, where m is an application or group message:
$\exists x : x =$*failure*: $(m.send \prec x \prec RRS[j]) \Rightarrow (RRS[k] \not\prec m.receive)$

Proof: If $\exists x : x =$failure: $m.send \prec x \prec RRS[j]$ then $m.send.f[j] < RRS[j].f[j]$. 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 before x. From R3 for P_i, for all $l, RRS[l].f = RRS[i].f$. That is, P_i

requests replies until all replies have the latest information on incarnations. These conditions lead to $RRS[k].f[j] = RRS[j].f[j]$ and $m.send.f[j] < RRS[k].f[j]$. Then, from SR2 and SR5, any group or application message received after $RRS[k]$ will be discarded, therefore, $RRS[k] \not\prec m.receive$. $\Box$

The next lemma will be used to demonstrate that orphan states will not be created.

Lemma 5.4 *For any state s, failure state x, and application message m:*
$s \prec x \wedge s \rightarrow m.receive \wedge m.receive \in run \Rightarrow s$ *will be recreated.*

Proof: Let x be at P_i. Assume that for any previous failures on P_i before s, the lemma statement holds. We will show that the lemma continues to hold for all s after these failures but before x. We prove by considering all types of events which can occur. Let $m.receive \in run_j$ (that is, m is received at P_j) and RRS be the recovery reply state corresponding to failure x.

1. s immediately precedes a send or internal event – Since we are considering deterministic execution, s will be recreated if all receives which precede it are recreated.

2. s immediately follows a receive and P_j is outside of P_i's group – A determinant is created for the receive before s and placed in $\mathcal{D}_i$ (by lines immediately following delivery in SR5). To send m, (by SR1) all such determinants which are not already at $K+1$ processes will be sent as a group message. Upon receiving K log-ack messages for this send, all determinants at the sender of m will now be at $K+1$ processes. For all receives which precede $m.send$, there are now determinants at $K+1$ processes. When P_i fails, at least one process must exist which will not fail before sending the determinant for the receive to P_i. This is due to our limits on the number of simultaneous failures. P_i will receive the determinant because it must receive a recovery reply from all other processes within the group. Using lemma 5.1, s will be recreated, because the receive will be recreated.

3. s immediately follows a receive and P_j is within P_i's group – there are two cases:

   (a) $m.receive \prec RRS[j]$ – there are two subcases:

      i. $\nexists y : y =$ failure $\wedge m.receive \prec y \prec RRS[j]$ – Since there does not exist a failure between the receive and the RRS, all determinants known by P_j at $m.receive$ will have been sent to P_i in a recovery reply message. By invoking lemma 5.1, s will be recreated.

      ii. $\exists y : y =$ failure $\wedge m.receive \prec y \prec RRS[j]$ – Again, there are three subcases, where m_0 is any message which may affect causality (these are application messages and recovery reply messages):

         A. $\exists m_0 : m.receive \rightarrow m_0.receive$ and $m_0.receive \prec RRS[m_0.dest]$ without an intervening failure – P_i will receive determinant information for s because $m_0.receive \prec RRS[m_0.dest]$. Using lemma 5.1, s will be recreated.

         B. $\exists m_0 : m.receive \rightarrow m_0.receive \wedge m_0$ is received outside of P_i's group – Let $m_0.receive$ be $m.receive$ and consider case 2. $s \rightarrow m_0.receive$ and s is recreated.

         C. Neither of the previous two conditions apply – Then 1) there does not exist a message sent between $m.receive$ and y or 2) all messages (application and recovery replies) which were sent causally following $m.receive$ are within the group and were sent to processes which failed after this send but before RRS. In 1), $m.receive \notin run_j$ because no message is sent from $m.dest$ which would preserve the fact that it has been received. In 2), the same is also true because all messages which causally follow m have intervening failures between their receive and RRS. Therefore, the information preserving $m.receive$ no longer exists, so $m.receive \notin run_j$.

(b) $RRS[j] \prec m.receive$ – For this case, there cannot be a failure between $m.send$ and RRS (or m would not be delivered, by lemma 5.3). This implies that the process which sent m received a message before $m.send$. The proof for case (a) can be followed.

□

We now show that our algorithm satisfies the properties of consistency and safety given earlier.

Theorem 5.5 *The group message logging algorithm satisfies the property of consistency.*

Proof: Recall the property of consistency:
All states which precede non rolled back states are also not rolled back.

$$\forall s, t \in S : t \in run_{\mathcal{P}} \wedge s \rightarrow t \Rightarrow s \in run_{\mathcal{P}}$$

By the previous lemma,

$$\forall s \in S, m \text{ an application message} : m.receive \in run \Rightarrow s \in run$$

This leads to our result since for all t between $m.receive$ and the next receive, t will be recreated due to deterministic execution. □

Theorem 5.6 *The group message logging algorithm satisfies the property of safety.*

Proof: Recall the property of safety:
A process will not perform output commit unless it is recoverable:
$\forall s, m : (s = \text{state preceding send of } m) \wedge (s \in execution_i) \wedge (m = \text{commit message}) \Rightarrow s \in run_i$
Lemma 5.4 shows that any message received outside of a group will not be removed from the run. Therefore any output commit send will remain in the run, since $m.send \rightarrow m.receive$. This leads directly to the property of safety. □

Theorem 5.7 *The algorithm is deadlock free.*

Proof: We need to show that sends outside of a group and recovery can always complete. Since only K log-acks are required to send, and groups are at least $2K+1$ in size, sends are never blocked (because only K failures can occur within a group). Since all processes can eventually respond with recovery replies, whether recovering or not, any process can recover. □

6 Complexity

Our algorithm requires that a process wait for K acknowledgements before sending output commit. As stated earlier, ring message logging requires $2N$ message hops. Other approaches require synchronous writes to stable storage which we show to be slower.

We require the overhead of additional messages for our algorithm. This is $O(K)$ messages for messages sent outside of a group. Half of the extra messages are empty, the other half contain only determinant information. While this is better than ring message logging, which requires $O(N)$ messages, it is worse than other message logging approaches that do not have additional messages. However, by keeping most communication within groups, these extra messages can be minimized.

For every failure that occurs, only one process must roll back in our scheme. This is similar to causal and pessimistic message logging, and better than optimistic logging which can cause $O(N)$ processes to roll back.

7 Experimental results

We implemented our algorithm to gather statistics on the performance for output commit. For our experiments we configured processes into groups of size $2k + 1$ and varied k from 1 to 8. We then timed from before the send of the Group messages to the receive of k Log-Ack messages. We performed 10 runs under midday network traffic for each value of k and averaged the results. Non-blocking interrupt driven datagram sockets were used to best reflect use in a real program. Every process was placed on its own host. These hosts consisted of Sun Sparc 10/40s, RS6000 350 and 34H machines, and Sun 4s. All times were collected on Sun Sparc 10/40s. The network connection was 10BaseT (10 Mbits/sec.) The timings for our algorithm for output commit are given in figure 16. For comparison to the standard output commit approach for causal message logging, we timed the write of the determinant log to a single hard disk on the local host. Two times are given on the graph, the lower value is for disk writes with zero contention, while the higher is when disk contention occurred approximately 10% of the time, where the conflict was large file writes. The latter is merely to show how contention can cause large waits. The write timings were gathered on a Fast-Wide SCSI disk with 9ms seek. We performed 100 log writes and averaged the results. The logs were not concatenated because this might cause additional seeks as the file grew larger. We note that if past trends continue, the gap betweens the times for our algorithm and those for writing to disk will widen[1].

8 Conclusions

We have presented a recovery algorithm that reduces the latency of output commit. This is done by sharing receive information using send and acknowledge messages and breaking the system into

[1]The reference Sparc 10/40 machine has SPECint95 and SPECfp95 numbers of 1 and 1. Current (1997) low end workstations have numbers of 3 and 5 while high end workstations reach 10 and 18 (An Intel Pentium-Pro 200 is around 8 and 6). Newer 100BaseT networks transfer data at 100Mbits/sec. It should be noted that processor and network performance gains continue to outpace gains in hard disk seek times. Our 1995 drive, has a seek time very near a 1997 high end drive of 7.5msec. This trend is predicted to continue in the future.

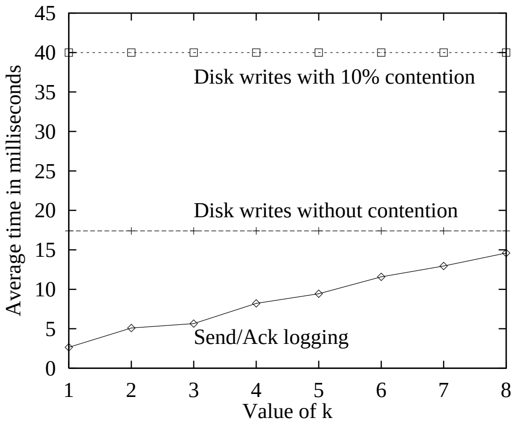

Figure 16: Comparison of delay required for output commit for group message logging and logging to stable storage (disk).

groups to keep the number of acknowledgements small. Those groups which tolerate one failure do not have to wait for an acknowledgement before sending outside of a group (see appendix). The additional messages we use are small (sometimes empty) and the response to these can be at a low level, rather than at the application level. We have presented experimental results that show that for groups with $k = 2$ or 3 our approach is over three times faster than simply writing to stable storage. Our approach is an improvement over other methods for fast output commit, which require a write to stable storage or $2n$ message hops. Our recovery method does not create orphan messages, unlike optimistic and ring protocols, and tolerates a greater number of failures, in general, than family based message logging. Finally, our implementation of family-based logging never blocks live processes and allows for independent recovery during simultaneous failures, something previous work has not accomplished.

References

[1] L. Alvisi, B. Hoppe, K. Marzullo, "Nonblocking and orphan-free message logging protocols," *Proceedings of the 23rd Fault-Tolerant Computing Symposium*, June 1993, pp .145-154.

[2] L. Alvisi, K. Marzullo, "Message Logging: Pessimistic, Optimistic, and Causal," *Proceedings of the 15th International Conference on Distributed Computing Systems,* IEEE Computer Society, May 1995, pp.229-236.

[3] 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.

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

[5] 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.

[6] F. Cristian, F. Jahanian, "A timestamp-based checkpointing protocol for long-lived distributed computations," *Proceedings of the 10th Symposium on Reliable Distributed Systems*, Sept. 1991, pp.12-20.

[7] G. Chiu, C. Young, "Efficient Rollback-Recovery Technique in Distributed Computing Systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, no. 6, June 1996, pp.565-577.

[8] O. Damani, V. Garg, "How to Recover Efficiently and Asynchronously when Optimism Fails," *Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996, pp.108-116.

[9] 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.

[10] E. Elnozahy, W. Zwaenepoel, "On the Use and Implementation of Message Logging," *Proceedings of the 24th International Symposium of Fault-Tolerant Computing*, 1994, pp.298-309.

[11] 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.

[12] P. Fingar, J. Stikeleather, "Distributed Objects for Business: Getting started with the next generation of computing," a whitepaper adapted from *Next Generation Computing: Distributed Objects for Business*, P. Fingar, D. Read, J. Stikeleather, SIGS Book & Multimedia, New York, 1996.

[13] P. Leu, B. Bhargava, "Concurrent robust checkpointing and recovery in distributed systems," *Proceedings of the 4th International Conference on Data Engineering*, Feb, 1988, pp.154-163.

[14] B. Lampson, H. Sturgis, "Crash recovery in a distributed data storage system," XEROX Palo Alto Research Center, April 1979.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

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

9 Appendix

When $K = 1$ the algorithm can save one message hop during sends outside of a group. To send outside of the group requires the message and send log be forwarded to another process in the group. This process then forwards the message to the destination, and sends a log-ack message to the sender. The idea is that in most cases the process will operate correctly and return a log-ack. If it fails before sending the message or the log-ack, the absence of a log-ack eventually will cause the sender to resend the message. Since messages are numbered, if one is actually sent multiple times, the destination process can discard the extra messages.

The send/receive rules for processes in a group where $K = 1$ are given in figure 17. A description follows:

- **SR1**: *sending an application message* – increment the count of the number of messages sent, and assign to message. If the destination of the message is within the group then send the send log (determinants and incarnation) with the message. If the destination is outside of the group and the message can be sent without waiting (because no new messages have been received since the last send), send to destination. Otherwise, choose a process and send determinant and message to it.

- **SR2**: *receiving a group message* – combine determinants from message with those stored for sender. If the message has the current incarnation number, send the application message to destination and log-ack to sender.

- **SR3**: *receiving log ack message* – increment num_acks_m of message.

- **SR4**: *if a log ack has not been received for m* – send m again.

```
upon message m to send to P_j do                                              SR1
    msgs_sent_k = msgs_sent_k + 1;
    for all (D ∈ 𝒟_k : D.h ≤ k)
        add D and message for D (if present) to 𝒟_m;
    if (P_j within group) then
        send (f_k[i], 𝒟_m, msgs_sent_k) with m as application message to P_j;
    else
        send (f_k[i], 𝒟_m, msgs_sent_k) with m as group message to another
            process within group;
    save m, d_m = P_j, and n_m = msgs_sent_k for sending/resending;
upon receiving group message from P_j and P_k not recovering do               SR2
    remove (f_m, 𝒟_m, n_m) from m;
    if (f_m = f_k[j]) then
        send m to d_m;
        send log-ack_m to P_j for m;
        get_determinants(𝒟_m);
upon receiving log-ack_m do                                                   SR3
    num_acks_m = num_acks_m + 1;
if ∃m : num_acks_m = 0 and waited too long then                               SR4
    send m again;
upon receiving application message, m, from P_j and P_k not recovering do     SR5
    if (P_j ∈ 𝒫) then remove (f_m, 𝒟_m, n_m) from m;
    if ((P_j within group and f_m = f_k[j]) or (P_j not in group) then
        msgs_rcvd_k = msgs_rcvd_k + 1;
        get_determinants(𝒟_m);
        deliver m;
        𝒟_k = 𝒟_k ∪ (n_m, P_j, P_k, msgs_rcvd_k, 1);
```

Figure 17: Send/receive rules for P_k with $K = 1$.

- **SR5**: *receiving application message* – if message from within group and incarnation number is correct: save determinant and deliver message, otherwise deliver message. Finally create a determinant for the receive, and save.

Note that the properties used in proving lemmas 5.3 and 5.4 also hold for $K = 1$. That is, a failure does not cause events to be orphaned. Therefore, the algorithm for the special case of $K = 1$ satisfies the properties of safety and consistency.