Sharing state information for fast recovery in a k faulty system

J. Roger Mitchell* Vijay K. Garg†

Parallel and Distributed Systems Laboratory
http://maple.ece.utexas.edu
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712

March 14, 1996

Abstract

There have largely been two approaches to recovery in a distributed system following a failure: checkpointing to stable storage and replication of processes. Both allow a process to fail and recover assuming stable storage or a replicate is not adversely affected during the failure. In the approach presented here, we use checkpointing at another process which does not require either replicates or stable storage. This provides the benefits of lower communication overhead as compared to replicates and faster recovery as compared to stable storage.

1 Introduction

In a distributed system as in life, few things lasts forever. Because of this, in any distributed computation that is run long enough, one of the processes is sure to fail. This failure can be due to software error, hardware failure, or some outside influence. The recovery method presented in this paper addresses these failures. Specifically, this method employs checkpointing and roll back of the failed process. Typically checkpointing involves a process saving its current state, so that in the event of failure, most work is not lost. This checkpoint can be stored in stable storage or at some other process. Our approach is to store this checkpoint at another process of the computation. This provides two benefits over previous approaches: faster recovery and lower communication overhead.

*supported in part by MCD and Virginia & Ernest Cockrell fellowships

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

Recovering a checkpoint from stable storage can often be costly. Not only can the time to recovery be long (when compared to processor speeds), but the last checkpoint stored may also be one which is quite old given the latency involved in storing checkpoints to disk. This can mean that the recovering process may have to reexecute much of its run just to reach the point where it failed. Using stable storage for recovery also requires that execution be deterministic. This is because checkpoints are stored in the background, meaning they are always of process states that are old. Therefore, a log must be maintained of messages received since the last checkpoint. Following a failure and recovery of the checkpoint, the log is replayed, bringing the process back to the state it was in immediately before the failure. This is called *message logging*.

Typically, two types of message logging are considered: *pessimistic* and *optimistic*. In pessimistic message logging, [4, 15] when a message is received, it is saved to stable storage in a log and the process is blocked from sending or receiving any more messages until this operation is complete. During recovery the failed process reads its last checkpoint and then begins execution by reading logged messages. Because the execution is deterministic, messages sent during this replay of logged messages will be duplicates of those sent before the failure. These messages are then ignored by a receiving process. Optimistic message logging avoids the delay caused by the pessimistic approach by logging messages to stable storage in the background while continuing normal execution. When a failure occurs however, those messages not logged are lost. Any events in other processes that causally follow these lost receive events are called *orphan events* and must be rolled back.

Another approach to recovery is to replicate processes or tasks. Primary-backup replicates tasks (typically a request due to an RPC) by performing the request on a backup when the primary fails. This requires reexecution of the task. By replicating a process, when the primary fails the replicas can take over. In normal execution the replicas can be kept up to date by performing the same execution as the primary (active replication), or informed of changes to the primary's state (passive replication). In either case, implementations to do this have large communication overheads. In addition, processor utilization is higher due to extra process overhead.

2 Previous work

Recovery using message logging was first presented using the pessimistic approach [4, 15]. Because this approach blocks execution and failures tend to be rare, later work has generally been in optimistic logging. The first paper using optimistic logging was by Strom and Yemeni [20]. Their approach, however, may cause an exponential number of roll backs at processes, due to the *domino effect*. Johnson and Zwaenpoel's [10] algorithm avoids these exponential roll backs by calculating the maximum recoverable system state by requiring synchronous recovery. Sistla and Welch's approach [18] also minimizes roll back but requires FIFO ordering of channels and vector clocks. Smith, Johnson, and Tygar [19] further refine the optimistic approach by providing a recovery method that is asynchronous. This method, however, requires large timestamps. Damani and Garg [6] improve upon this work by presenting an algorithm that has smaller timestamp requirements. The problem with optimistic message logging is that orphan events can be created. This means that the failure of any one process may require all processes to roll back, so that the system can return to a consistent state. To avoid orphan events, Alvisi and Marzullo[2, 1] discuss a method of message logging in which the message log and its ordering is maintained by other processes. For each message, a *determinant* is created which consists of the source, send ordering, and receive ordering. It is assumed that with a determinant, the sending process can reconstruct the message. The determinant is stored at the receiver which includes the determinant in any message it sends. This action spreads the information across the system and ensures that any message that causally precedes its own message send will have its determinant included in the message. Therefore, the receiver of any message receives determinants in causal order. The number of processes that store duplicate determinants is based upon the number of possible simultaneous failures. This approach is called *family based* message logging. In the failure of a single process, however, all processes may have to become involved in piecing together the message log at the recovering process, and like all message logging approaches, the checkpoint must be retrieved from stable storage.

There have been many approaches to recovery using replication. The SIFT, MAFT and MARS

[12] real-time distributed architectures use active replication, and have specialized hardware requirements, such as synchronous environments, that do not lend themselves to comparison. Delta-4 [14] was also developed for real-time environments. The Delta-4 literature presents three types of replication: *active, semi-active, and passive.* Active and semi-active require that replicas at least partially perform the same execution as the primary. Passive replication, used by Delta-4, requires that the replicas receive all messages directed to and from the primary, as well as receiving state updates with every message sent. This means that execution does not have to be deterministic. However, with a replica group of size k, whenever a message needs to be sent from the primary, a protocol is invoked which is $O(k^2)$. In addition, this protocol requires the use of an underlying atomic broadcast protocol [1] as well as timeouts under certain failure types. The ISIS system [3] also can use any of the three types of replication, in general. Passive replication is used with replicas called coordinator and cohorts. These coordinating processes are part of an RPC object, similar to primary-backup, and when the coordinator fails, a cohort takes over. The approach requires $O(k)$ messages to send, in addition to an underlying atomic broadcast mechanism (here called BCAST). Higaki and Soneoka [8] also present a method for passive replication. This *group-to-group* communication also implements an atomic broadcast and requires $O(k)$ send messages similar to ISIS.

Another approach to recovery is synchronous checkpointing. Ramanathan and Shin and Randell, et. al. [16, 17] call this method a *conversation scheme.* In this approach, some process initiates a conversation among a group of processes so that a checkpoint can be taken by each process which are then guaranteed to be concurrent. These concurrent checkpoints are the furthest any process will have to roll back. The major drawback of this approach is the necessity for synchronous checkpointing and, because of this, wasted execution time as fast processes wait for slow processes.

Figure 1 gives a comparison of the method presented in this paper with the work just described. The work in this paper is called agent/customer recovery (ACR). In the figure, m is the number

[1] Atomic broadcast requires that all correct processes receive all messages in the same order.

Algorithm/ System	Message ordering	Number of msgs	Msg over- head	Msg hops	Max roll- backs per fail	Max fails before cannot recover	All HW fails allowed	# of chkpts / proc	Recovery time
Ramanathan Shin	Synch	$O(m)$		1	n	n	No	1	αS
Johnson Zwaenepoel	None	$O(m)$	$O(1)$	1	n	-	No	1	αS
Peterson Kearns	None	$O(m)$	$O(n)$	1	n	1	No	1	αS
Strom Yemeni	FIFO	$O(m)$	$O(n)$	1	$\Theta(2^n)$	1	No	1	αS
Sistla Welch	FIFO	$O(m)$	$O(n)$	1	n	1	No	1	αS
Smith Johnson Tygar	None	$O(m)$	$O(n^2k)$	1	n	n	No	1	αS
Damani Garg	None	$O(m)$	$O(n)$	1	n	n	No	1	αS
Alvisi Marzullo	FIFO	$O(m)$	$O(n)$	1	1	k	No	1	αS
Delta-4	Atomic b'cast	$O(k^2m)$	State	2+ b'cast	1	k	Yes	$O(k)$	αM
ISIS	Atomic b'cast	$O(km)$	State	2+ b'cast	1	k	Yes	$O(k)$	αM
ACR (our algo.)	None*	$O(km)$	State	2 or 3	1	k	Yes	$O(k)$	αM

*detection of out-of-order checkpoint messages required.

Figure 1: Comparison of recovery methods

of application messages, k is the level of fault tolerance desired, and n is the number of processes in the system. Under message overhead *state* means that the process state must be included in some messages. The ISIS and Delta-4 entries indicate that the number of message hops includes any overhead associated with the required broadcast. Under the "recovery time" heading, αS means proportional to stable storage speed, whereas αM means proportional to volatile memory speed. In general, there is division between those algorithms/systems that use stable storage for checkpointing and those that use volatile storage. When checkpointing is done on stable storage it is assumed that this checkpoint is in the same physical processor box. Precluding the use of special hardware, this makes a process recovery vulnerable to a single failure of this processor. When using volatile storage for checkpointing, the system is only vulnerable following failure of storage of all k copies of the checkpoint. These points are listed as "HW fails allowed." Checkpointing to volatile storage also means the speed of recovery is proportional to the access time of volatile storage (RAM) rather than stable storage (magnetic media).

The ACR method presented in this paper compares favorably to previous work. The advantages of our approach over stable storage checkpointing are faster recovery time, greater flexibility in amount of fault tolerance provided, and greater tolerance of hardware failures. Rather than being dependent on the speed of stable storage, ACR is dependent on memory and network speed. As networks become increasingly fast, this speed advantage should increase. The advantages do come with some cost. As figure 1 indicates, the size of messages are larger in ACR, message latency is longer, and there are more messages. The number of message hops required in ACR is 2 for $k = 1$, and 3 for $k > 1$. This is better, however, than Delta-4 and ISIS. The advantages of our approach over volatile storage checkpointing are lower communication overhead, and smaller process overhead. As the figure indicates, message overhead is linear with respect to k, and there is no underlying message ordering protocol. The number of message hops in ACR is smaller or equal to Delta-4 and ISIS, which require synchronizing broadcast protocols with each send. Finally, like these two systems, and family-based logging, the ACR algorithm can be tailored to the number of failures the designer wishes the system to tolerate simultaneously, and does not require that nonfailed processes roll back. Like Delta-4 and ISIS, ACR can also use a non-deterministic execution model.

In the rest of the paper examples will be given and correctness presented. We will show that the ACR method efficiently provides desireable properties of a recovery algorithm.

3 An example with a two process group

Suppose there is a system of N processes, in which processes 1 and 2 are the focus. Processes 1 and 2 act as a group. Assume, for now, that each shares its current state with the other every time a message is sent. To do this, all messages sent from P_1 are routed through P_2 and vice versa, with state information piggybacked on the message. When a message is routed through either of the processes in this way, the routing process removes the current state information from the sending process and sends a forwarded message. The process that saves the state information and sends the forwarded message is called the *agent* and the sender of the original message is called the *customer*. Obviously, it is desired to have the customer and agent as geographically close as possible, ideally in the same building, or room. However, to allow for hardware failures, they should be on distinct physical processors. The following message definitions will be used for the presentation:

- *Customer message*: A message sent by a customer to its agent which includes state information for the customer.
- *Forwarded message*: A message resent by an agent after removing and saving state information from the original message.
- *Recovery request message*: The message sent by a failed process to its agent requesting recovery information.
- *Recovery reply message*: The message containing state information for recovery sent by an agent to its customer.
- *Retransmit message*: A message sent by a recovering process telling other processes to resend lost messages.

An example of how a failure within the group, P_1-P_2, is handled, follows. Figure 2 shows the activity for P_1 and P_2 where P_1 is acting only as a customer and P_2 only as an agent. To simplify the example, process P_2 is not performing any application activity. Process P_1 sends a message at a, a customer send. P_2 performs an agent receive and then an agent send, forwarding the message to its proper destination. P_1 during this time has also received message $m1$ from another process. Following this receive, P_1 fails, and because P_1's last message send was at a, this is effectively the last checkpoint taken for the process. Eventually P_1 is restarted and sends a recovery request message to P_2 indicating it failed and asking for its stored state information. P_2 responds by sending the state information within a recovery reply message. Following receipt of this message, P_1 restores state a, sends a retransmit message to all sending processes, and resumes execution.

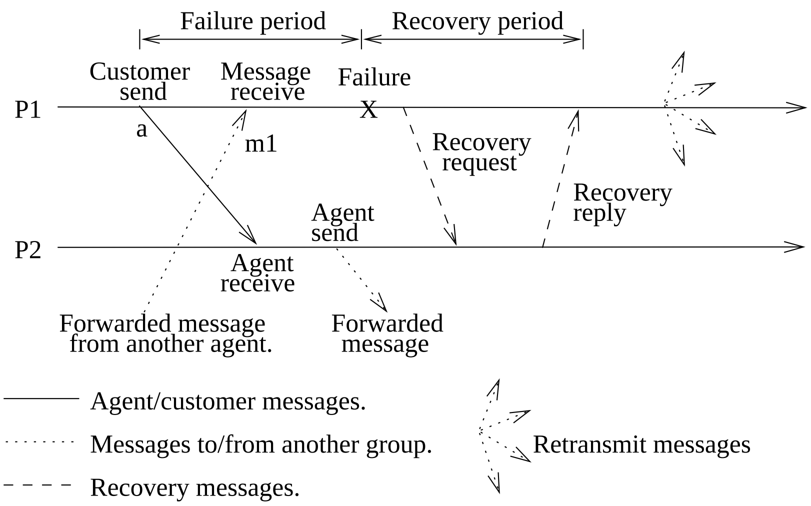


Figure 2: A failure in a two process group

The retransmit messages from P_1 include a list of all messages received from the corresponding destination. A process that receives a retransmit message compares what it has sent to what is indicated and resends the necessary messages. Some of the events are defined as follows, and also labeled in figure 2.

- *customer send*: The send of a message from a customer to its agent.
- *agent receive*: The receive of a message from a customer at the agent.
- *agent send*: The send of a forwarded message.
- *message receive*: The receive of a message at its final destination.
- *failure period*: The period of any process execution from failure back to the last send before the failure.
- *recovery period*: The period from failure until recovery information is received from the agent.
- *simultaneous failures*: Failures which are concurrent or for which one failure occurs after the receive of the recovery request message from the other failure and before the send of the corresponding recovery reply message.

The example can be expanded to a group of size 3 (or greater). With three processes in a group, each process will have one agent and one *assistant*. The customer would send state information to both types of agent but the message only to the agent. The actions of the assistant become clear in the algorithm specification and section 5.4.

It is immediately apparent that this approach requires $O(km)$ total messages where m is the number of application messages and k is the size of the group. Since all but m of these are within a group, the members of which are in close physical proximity, the extra overhead is contained within the group. The number of extra messages is in fact smaller than the methods employing passive replication: ISIS, Delta-4, and group-to-group communication. Also, the ACR method requires that every agent store an up-to-date state, or checkpoint, of its customers. Passive replication also has this requirement.

4 The Algorithm

4.1 Distributed environment

In this paper, we assume that processes fail by ceasing all activity. We also assume timely detection of a failed process, reliable channels, and that the agent recognizes out of order messages. With knowledge of out of order messages, the agent can keep the most current state variables stored[2]. It is also assumed that during a recovery period a process buffers any messages received. Finally, the checkpoint maintained by an agent is capable of fully reinitializing the failed customer by our definition. Therefore, the checkpoint includes lists of messages received and actual messages sent. In the "Deterministic execution and other considerations" section, methods to keep these lists small are given.

[2]A discussion of this will not be given in the main paper, instead see the appendix for examples and methods for implemention.

4.2 Notation

Events at a process are represented using small letters, for example e and f. The notation $e.p$ will mean the process on which event e occurred. The symbol '$\rightarrow$' will be used for Lamport's happened-before relation [13]. '$e \leadsto f$' will mean that e is a send event and f is the corresponding receive event on a different process. '$e \prec f$' will mean that event e happened before f on the same process, and '$e \preceq f$' will mean $e \prec f$ or $e = f$. The notation A_i will be used to signify the agent process for P_i and G_i the set of any other processes which are in the same group as P_i.

The following definition for correct recovery will be used.

Definition 4.1 *The correct recovery of a process after a failure satisfies the following:*

1. *The failing process is restored to a state, s, that occurred in its execution before the failure.*
2. *All messages received after s but before the failure are recovered.*
3. *Any messages sent after s but before the failure are recreated or all processes correctly rollback to points before receiving these messages.*

4.3 Algorithm specification

Figures 3 and 4 give the process code necessary to implement the recovery algorithm for any group size.

As indicated in the introduction, the algorithm uses *customers* and *agents*. Each application process acts as both a customer and agent. In the most general approach, every two application processes are paired together and each is an agent and customer of the other.

The duties of an agent are to 1) store state information for its customer (that is, a checkpoint) 2) forward messages received from its customer to the final destination after all processes in group have received the state information and 3) send state recovery information to its customer in the case of the customer's failure. As shown in figure 3, these tasks are implemented with A1- A5. The counter *failure* is used to distinguish between customer messages from different execution periods. The customer receives *failure* with the state variables during recovery and attaches this number

SV = State variables.
failure = number of failures of customer.
Agent(i): /* agent or assistant. */

upon receive(m, sv) **do**
    if $m.failure$ =failure **then** SV = update_state_vars(SV,sv);   A1
    /* update SV to reflect information from customer */
    if $m.dest \neq i$ **then do**
        if $m.agent \neq i$ **then** send Ack to $m.agent$;   A2
        if $m.agent = i \wedge$ Ack received from assistants **then**
            send(m) to $m.dest$;   A3
    else if $m.agent = i \wedge$ Ack received from assistants **then**
        deliver(m);   A4
upon receive(recover_request) **do**
    failure++;
    send(SV, failure) to $customer$;   A5
        /* This is the recovery reply message. */

Figure 3: Agent and assistant code for processor i

to all messages sent to the agent following this recovery. When the agent receives a message with *failure* less than its current value, the state information is not updated.

The duties of the customer are to 1) append state variables to all outgoing messages and route these through the agent, 2) send state variables to other processes in the group, 3) send a recovery message to the agent after failure and restore itself from a recovery message once received, 4) send retransmit messages after recovery, and 5) send lost messages in response to a retransmit message

Customer(i):

upon message m to send **do**
    sv = changed state variables for i;   C1
    send(m, sv) to $agent$;   C2
    send(sv) to all other processes in group excluding $agent$;   C3
upon recognize_failure send(recover_msg) to $agent$;   C4
upon receive(SV, failure) **do**
    recover(SV);   C5
    send(restransmit_msg) to all sending processes;   C6
upon receive(retransmit_request) **do**
    check retransmit_list for missing/last messages and resend;   C7

Figure 4: Customer code for processor i.

and in addition the last message sent if the retransmit message is from the agent. Lines C1 - C7 implement these requirements.

4.4 The algorithm for one failure per group, (k=1)

The following lemmas and theorems apply to the case in which processes are in groups of size two, each the agent of the other within the group. In this case, the algorithm does not allow for simultaneous failures in a group. Later it will be shown that the group size can be increased to allow for k simultaneous failures per group. The following are given to help in the analysis of the algorithm:

msg(e)	$\equiv$	the set of information sent or received at event e.
state(e)	$\equiv$	the state of process $e.p$ immediately after event e.
checkpoint(i)	$\equiv$	the set containing state(e)'s held by P_i.
failed(e)	$\equiv$	e is within the failure period of $e.p$.
lost(e)	$\equiv$	$\exists f : e \prec f \wedge$ failed(f): $\forall i$: state(e) $\notin$ checkpoint(i).
customer_send(e)	$\equiv$	e is a send to agent and state(e) $\in$ msg(e).
agent_rcv(e)	$\equiv$	customer_send(e) $\wedge \exists f : e \rightsquigarrow f$.
agent_send(f)	$\equiv$	$\exists e : e \rightarrow f \wedge$ agent_rcv(e) $\wedge g$ is send: msg(g) $\subset$ msg(f).

The first lemma demonstrates that any event that locally precedes a customer send can never be lost.

Lemma 4.2 *For all events e, f: $(e \preceq f) \wedge$ agent_rcv(f) $\Rightarrow \neg$lost(e).*

Proof: Let h be the current event such that $f \prec h$.

1. failed(h):

   failed(h) $\wedge$ agent_rcv(f) $\Rightarrow$ $\forall e \preceq f$: state(e) $\in$ checkpoint($A_{f.p}$)
   {by definition of agent_rcv, by line A1,
   and because $A_{f.p}$ cannot fail since $f.p$ has.}
   $\Rightarrow$ $\neg$lost(e)

2. $\neg$failed(h) $\Rightarrow \neg$lost(e) {by definition of lost.}

$\square$

An *orphan* event is an event which causally depends on some lost event in a failed process. Orphan events cause rollbacks at processes other than the failed process.

$$\text{orphan}(e) \equiv \exists f : \text{lost}(f) \wedge (f \rightarrow e) \wedge (f.p \neq e.p)$$

$$\text{rollback}(e) \equiv \text{lost}(e) \vee \text{orphan}(e)$$

That is, e will be rolled back if it is lost or becomes an orphan. The lemma, theorem and corollary all demonstrate the limited affects a failure has on processes. Specifically, correct processes never roll back, and failed processes do so to a small degree.

Lemma 4.3 *For all application events e : $\neg$orphan(e).*

Proof: There are two cases:

$$\begin{array}{lll} 1)\ \nexists f : f \rightarrow e \wedge e.p \neq f.p & \Rightarrow \neg\text{orphan}(e) & \{\text{by definition of orphan.}\} \\ 2)\ \exists f : f \rightarrow e \wedge e.p \neq f.p & \Rightarrow \exists g : f \preceq g : \text{agent\_rcv}(g) & \\ & \Rightarrow \neg\text{lost}(f) & \\ & \Rightarrow \neg\text{orphan}(e) & \end{array}$$

□

The next theorem states that events not lost will never be rolled back.

Theorem 4.4 *For all application events e : $\neg$lost(e) $\Rightarrow$ $\neg$rollback(e).*

Proof:

$$\begin{array}{ll} \text{For all application events } e : \neg\text{lost}(e) & \Rightarrow \neg\text{lost}(e) \wedge \neg\text{orphan}(e) \\ & \qquad \{\neg\text{orphan}(e) \text{ is a tautology, by Lemma 4.3}\} \\ & \Rightarrow \neg\text{rollback}(e). \\ & \qquad \{\text{definition of rollback}\} \end{array}$$

□

Because application events are never lost on unfailed processes, Theorem 4.4 and Lemma 4.2 lead to the following.

Corollary 4.5 *For all application events e : $(e \prec f) \wedge (f \leadsto g) \wedge$ agent_rcv(g) $\Rightarrow \neg$rollback(e)*

This corollary states that events on a customer preceding a customer send are never rolled back if an agent receive occured. It also becomes obvious that failures at one process do not cause rollbacks at another process, since events on a correct process do not become lost.

The next lemma states that any lost event will be recovered, where recovery is defined:

recovered(e) $\equiv$ [$\exists f : f \rightarrow e \wedge \neg$lost($f$): state($e$) recreated] $\vee$ e is internal event.

Internal events are recovered automatically since these are created based upon receive and internal events.

Lemma 4.6 *For all events e: lost(e)* $\Rightarrow$ *recovered(e)*

Proof:
lost(e) $\Rightarrow \nexists f : e \preceq f \wedge$ agent_rcv(f). (i)

1. e = receive event	$\Rightarrow$	(e is a *message receive*) $\vee$ (e is an *agent receive*) ($\exists g$: agent_rcv(g): $g \rightarrow e$) $\vee$ ($\exists g : g \rightarrow e \wedge \neg$lost($g$)) {for a *message receive* the message must have passed through an agent, for an *agent receive* the send from customer cannot be lost.}
	$\Rightarrow$	$\exists g : \neg$lost(g) $\wedge g \rightarrow e$ {by lemma 4.2, definition of agent_rcv, and simplification.}
	$\Rightarrow$	recovered(e) {g is not lost, so by (i), the message received at e can be resent. From algorithm, failed(f) $\Rightarrow$ recovery request $\Rightarrow$ recovery reply $\Rightarrow$ retransmit request for message received at e.}
2. e = send event	$\Rightarrow$	e is an *agent send* {by lemma 4.2, customer send events are never lost.}
	$\Rightarrow$	$\exists f$: f is an *agent receive* $\wedge f \prec e$
	$\Rightarrow$	recovered(f) {since f is a receive event: from case 1.}
	$\Rightarrow$	recovered(e) {e is the agent send event for the receive f.}
3. e = internal event	$\Rightarrow$	recovered(e) {by definition.}

□

Theorem 4.7 *The algorithm correctly performs recovery in the case of a single failure.*

Proof: The algorithm must satisfy the definition of correct recovery:

1. The failing process is restored to a valid point, s, in its execution before the failure.
2. All messages received after s but before the failure are recovered.
3. Any messages sent after s but before the failure are recreated or all processes correctly recover to points before any orphan events.

The first requirement is satisfied by considering corollary 4.5 because a failing process is restored to the state immediately following the last send event before the failure. This implies that no send events occur between s and the failure. In addition no orphan events are created (Lemma 4.3). Therefore, requirement three is satisfied. Requirement two is satisfied by the previous lemma. □

4.5 The algorithm for k simultaneous failures per group ($k > 1$)

The proofs and discussion presented so far have proven the algorithm correct for an agent/customer relationship in which there are only two processes. Since in this case, each process is the agent of the other, if both were to fail simultaneously neither could recover. To increase the number of allowable failures within an agent/customer group, the group can be expanded.

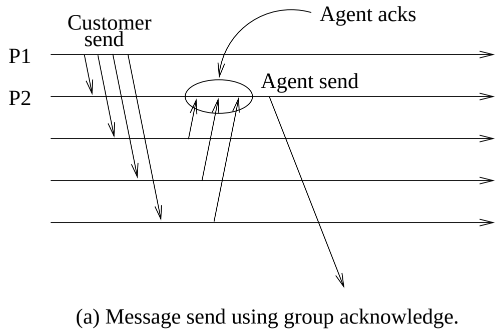
By expanding the group size to $k+1$, up to k simultaneous failures can be tolerated within the group. Since up to this point the paper has only discussed a *single* customer and a *single* agent, a group will be defined:

Definition 4.8 Group: *a set of processes such that if a customer process is within the group, then so are the agents for that customer.*

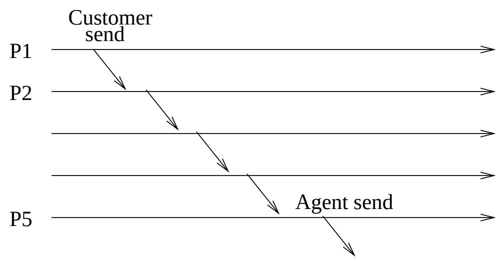
A group that is specified to tolerate a certain number of failures can also be defined:

Definition 4.9 k fault tolerant group: *a group for which up to k simultaneous process failures can occur and then all correctly recover.*

To create a k fault tolerant group, every process is assigned one unique process as its agent. That is, every process is only the agent for one other process. All processes which are not the agent of a process P_i are considered assistants of P_i. To implement the recovery algorithm for a group,



(a) Message send using group acknowledge.



(b) Message send by routing through group.

Figure 5: Two methods to send an application message from a group of size 5.

it is necessary that before an agent send event occurs, all agents in the group have received the state update associated with the message. This requirement is necessary for correctness, and can be enforced if the following is required of *agent_rcv*:

$$\text{agent\_rcv}(e) \equiv \text{customer\_send}(e) \wedge \exists g : g.p = A_{e.p} : \forall f.p \in G_{e.p} : e \rightarrow f \rightarrow g \wedge \text{ msg}(g) \subset \text{msg}(e).$$

Note that all the other definitions used for the case in which $k = 1$ continue to hold for $k > 1$. With the agent_rcv definition above, the proofs for the lemmas and theorems of the previous section are similar to those for $k = 1$. This leads to correctness by our definition for $k > 1$.

Figure 5 shows two methods of implementing the group send that satisfies the new agent_rcv definition. The message send of (a) matches the agent code of figure 3. Here, P1 as a customer sends a message with state update to its agent, P2. Next the state update is sent to all of P1's assistants. The assistants send an acknowledgement to P2 which waits until all have been received before performing an agent send. The message send of (b) would require a slight change to the agent code of figure 3. However, the predicates of section 4.3 remain valid, ensuring correctness. It is presented here as an alternative approach. In (b), P1 as a customer sends a message with state update to its agent, P2. This is forwarded through all assistants. When the message reaches the last assistant, this assistant, P5, performs the agent send. Note that in (b) the definition of an agent is slightly altered, because the duty of forwarding a message is given to a assistant. It is apparent then from both (a) and (b) that the message complexity of the algorithm requires $2k$ messages for every application message sent from a k fault tolerant group. The size of k, then, should be kept as small as possible and in general k equal to 1 or 2 will be sufficient fault tolerance for most applications.

As mentioned, the algorithm of figures 3 and 4 uses the approach of (a). As line $C3$ in the customer code indicates, when process P_i intends to send a message via its agent, it first sends the message with state variables to its agent, and then the state variables to its assistants. Here, the agent has the same duties as in section 4.3.

As an example, suppose there was a group in which the system designer desired to tolerate

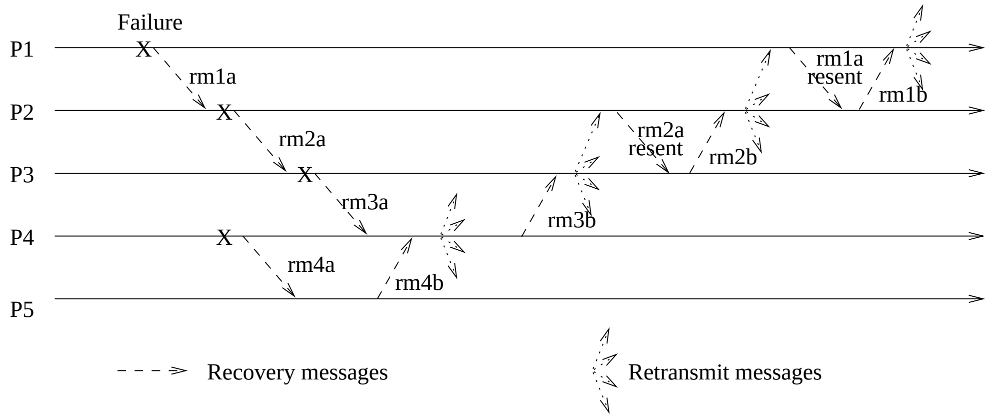


Figure 6: Four failures in a group with five processes

four failures simultaneously. The group would then consist of five processes. Figure 6 shows such a group, and the operation of the group in a case where four simultaneous failures occur. As the figure shows, at varying times, the four processes, P_1 through P_4 fail such that the recovery periods are at least partially concurrent. The failure of P_1 causally precedes the failures at P_2 and P_3 because of the recovery messages sent by the processes. Because of this sequence, both recovery messages $rm1a$ and $rm2a$ are lost. P_3 and P_4 also have failed and P_4 sends a recovery message to its agent, P_5. Meanwhile, P_3's recovery message is sent to P_4 but cannot be received because P_4 has not itself recovered. When P_4 receives recovery and state information from P_5 it receives all pending messages then reissues a retransmit message based on these messages. Following this, it recognizes that P_3 has failed and forwards information to it that it received from P_5. P_3 then sends a retransmit message. This is shown for process P_2 in the figure as event a. Once P_3 receives the recovery message, $rm2a$, for the second time, it replies with recovery information for P_2. Then P_2 and P_1 go through this procedure.

4.6 Considerations for group size, k

It is apparent from the previous section that although providing for large values of k in a group might lead to better fault tolerance, larger k leads to greater message delay and more messages.

In most cases, smaller k will typically provide sufficient fault tolerance. It should be noted that the algorithm provides for up to k failures per *group*. Therefore, different groups can be larger or smaller based upon the required fault tolerance. If, for example, a section of processes is extremely critical and requires recovery with a larger number of simultaneous failures, the processes in this area could be put into groups with large k while the rest of the system could be in groups with $k = 1$.

4.7 Complexity

As mentioned earlier, the message complexity of this algorithm is $O(km)$. The amount of saved data at each agent is $O(ks)$, where s is the size of the data or checkpoint for a single process. Because of these two facts, it is apparent that k should be kept to a minimum. For recovery, the number of messages sent for the process to begin reexecution is two. Since up to k simultaneuos failures can occur where each must recover in turn, the time to recover is $O(k)$. Once recovered, the process issues retransmit messages which are sent to all processes that can send messages to this process. Because all processes may also return lost messages, the total number of messages in this transaction is $O(n)$, where n is the total number of processes. It should be noted that the speed with which these messages are returned is not important, and does not hold up the execution of the recovered process. This is because the recovered process is not required to receive these messages at any certain time or in any particular order.

5 Deterministic execution and other considerations

The group sharing approach requires that a process send an update of its current state to its agent with each send of a message. Sending a complete checkpoint everytime could be prohibitively costly. There are two ways around this; which method used is dependent upon whether a determinstic or non-deterministic execution model is used.

If the execution model is assumed to be non-deterministic, then the actual checkpoint must be updated with each message send. To avoid sending the entire checkpoint, the change in the state of the customer could be computed since the last message send. Using this approach, the application would have to be one for which this sort of computation is efficient and not too costly. The initial checkpoint may be rather large in this case, as much of the data needed to reinitialize the customer is transferred. In some cases, however, subsequent state information may consist only of local variables. For an execution which is deterministic, the customer could use the above approach or only occasionally send the checkpoint and then during all other sends, include a list of messages received since the last send. The latter method would be a type of message logging. Of course, using this message logging approach would slow down recovery. In this case, the recovering process would be required to reexecute from the last checkpoint, replaying the message log as it did so.

The ACR algorithm as given in figures 3 and 4 does not address some special considerations. The first is in regard to lists of messages sent that are saved at each process. These are necessary at every process to respond to retransmit messages. To keep this list small, processes need to know when to purge messages. This can be done by sending the retransmit list whenever a message is sent. Then, a process can remove messages if the message is on the list. Retransmit lists can be kept small by having the first message number on the list be the highest message number for which all messages with numbers less than or equal to it have been received. The second consideration is in regard to processes which send very few messages. To avoid a long recovery period due to very old checkpoints, these processes could send periodic dummy messages to their agent which just included state update information. This is a consideration that must be made with all passive replication schemes.

Finally, in this paper we have not discussed a method for detecting failures. It is obvious that fast recovery is heavily dependent on the timely discovery of a process failure. The method chosen to be practical would probably be a timeout of an expected response. Unfortunately, even in the real-time systems presented as previous work, timely detection of all types of failure is a problem.

6 Conclusions

The ACR algorithm as compared to stable storage checkpointing provides faster recovery, greater flexibility in the level of fault tolerance (giving efficent use of resources), and greater tolerance of hardware failures. When compared to other volatile checkpointing methods, ACR provides lower communication and process overhead. Like some previous methods of recovery, ACR does not create orphan events, and therefore correct processes are not rolled back. By providing the ability to tailor the algorithm for k failures in a group, the ACR method can provide efficient use of execution and resources.

References

[1] L. Alisi, 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. Alisi, 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] K. P. Birman, "Replication and Fault-Tolerance in the ISIS System," *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, 1-4, Vol. 19, No. 5, December 1985, pp. 79-86.

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

[5] N. Budhiraja, K. Marzullo, F. Schneider, S. Toueg, "The Primary-Backup Approach," in S. Mullender editor *Distributed Systems*, ACM Press, 1993.

[6] O. Damani, V. Garg, "How to Recover Efficiently and Asynchronously when Optimism Fails," University of Texas at Austin, ECE PDS technical report TR-PDS-1995-014, August 1995. (Available at http://maple.ece.utexas.edu).

[7] E. Elnozahy, W. Zwaenepoel, "Transparent rollback-recovery with low overhead, limited rollback and fast output commit," *IEEE Transactions on Computers,* Vol. 41, #5, May 1992, pp.526-531.

[8] H. Higaki, T. Soneoka, "Group-to-group communications for fault-tolerance in distributed systems," *IEICE Transactions on Information and Systems*, Vol. E76-D #11, November 1993, pp. 1348-1357.

[9] D. Johnson, W. Zwaenepoel, "Sender-based message logging," *Digest of Papers: 17 Annual International Symposium on Fault-Tolerant Computing,* IEEE Computer Society, June 1987, pp. 14-19.

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

[11] T.Y. Juang, S. Venkatesan, "Crash recovery with little overhead," *Proceedings of the 11th International Conference on Distributed Computing Systems,* IEEE Computer Society, June 1987, pp.454-461.

[12] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, R. Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach," *IEEE Micro*, 1989, pp. 25-41.

[13] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, 21(7), July 1978, pp.95-114.

[14] D. Powell, ed., *Delta-4 – A Generic Architecture for Dependable Distributed Computing*, ESPRIT Research Reports, Springer-Verlag, 1991.

[15] 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.

[16] P. Ramanathan, K.G. Shin, "Use of common time base for checkpointing and rollback recovery in a distributed system," *IEEE Transactions on Software Engineering*, vol.19 #6, June 1993, pp.571-583.

[17] B. Randell, P. Lee, P. Treleaven, "Reliability issues in computing system design," *ACM Computing Surveys*, vol. 10 #2, June 1987, pp.123-165.

[18] 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.

[19] S.W. Smith, D.B. Johnson, J.D. Tygar, "Completely asynchronous recovery with minimal rollbacks," School of Computer Science, Carnegie Mellon University, 1994.

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