A Crash Recovery Technique in Distributed Computing Systems

Cheng-Ru Young and Ge-Ming Chiu

Department of Electrical Engineering and Technology
National Taiwan Institute of Technology
Taipei, Taiwan

Abstract

In this paper we propose a new mechanism for implementing checkpoint/rollback-recovery in a distributed computing system. A logical-ring structure is introduced for the maintenance of recovery-related information. Message processing order of a process is maintained by all other processes on its associated ring. It requires no time-consuming operations of writing order information into stable storage. As a result, fail-free overhead is small. When failures occur, only failed processes have to roll back to their latest checkpoints. Surviving processes continue execution without being blocked. Output commit is fast as it needs no synchronization before a message is sent to the outside world.

I Introduction

Rollback-recovery strategy is an attractive technique for providing fault tolerance to distributed applications in a system. It periodically records the state of a process in a stable storage, an action called *checkpointing*. In case failures occur in the system, processes can resume executions from their latest checkpoints, an act called *rolling back*. During normal operation, certain actions must be taken to ensure smooth recovery to a consistent state after a system crashes. It is important that a recovery mechnism does not incur excessive overhead while offering efficient rollback-recovery. In addition, the recovery function should be provided to applications in a transparent manner.

Checkpoint/rollback-recovery methods are normally categorized into three classes [4] – pessimistic message logging, consistent checkpointing, and optimistic message logging. Pessimistic message logging requires that each application message be synchronously logged on stable storage [1, 10]. Rollback-recovery is easy to implement. Fast output commit can be achieved. However, it costs a large amount of overhead during fail-free operation.

Consistent checkpointing methods require that the checkpointing actions be synchronized among all processes so that a globally consistent state is recorded in the checkpoints [2, 3, 9]. Crash recovery is easy and simple. When failures occur in the system, processes, including some fail-free ones, may have to roll back. Considerable latency may be resulted from additional synchronization actions required by an output commit [4].

In optimistic message logging methods, execution and communication of application processes are asynchronous. In addition, checkpointing actions are also taken in an asynchronous manner [5, 7, 8, 13, 15]. Fail-free overhead can be reduced as a result. However, surviving processes may have to roll back when failures occur. Output commit is delayed due to coordination among the processes.

A transparent recovery technique, which uses the concept of antecedence graph to maintain dependency relation among state transition intervals of processes, is proposed in [4]. It only requires limited rollback, and fast output commit can be achieved. Fail-free overhead is small due to asynchronous logging of messages. However, the antecedence graph that is added to each application message can be of considerable size. In addition, some portion of a process' antecedence graph can be repeatedly transmitted. Also the antecedence graph must be written to a stable storage.

In this paper, we propose an approach which can achieve most of the advantages described above. A concept of logical ring is introduced for the maintenance of information about message processing order. It incurs small fail-free overhead by asynchronously circulating small order messages around a ring. Order information need not be written to stable storage. It requires only failed processes to roll back to their latest checkpoints while surviving processes can continue execution without being blocked. Output commit is

1063-6927/94 $03.00 © 1994 IEEE

fast as it needs no synchronization before a message is sent to the outside world. The flexibility provided by our method allows the mechanism to be tuned to a system designer's needs.

II System Model and Assumptions

In a distributed system, processes communicate with each other by sending messages via the interconnection network. A process consists of a sequence of piecewise deterministic *state transition intervals* (or STI) [6, 8], which are initiated by processing incoming messages from other processes. Processes are deterministic in the sense that, if two processes start from an identical state, they produce the same result if they process an identical sequence of input messages. Furthermore, a process may send messages to other processes in an STI. As a result of such message passing, processes can become causal dependent on each other [15]. For example, figure 1 shows the execution of three processes, called p, q and r, in a distributed

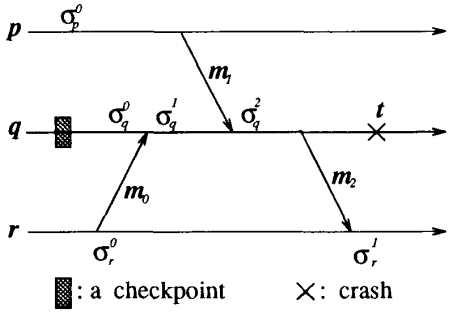

Figure 1: State transition intervals of processes.

system. The notation σ_p^i denotes the i-th state transition interval of process p. Process q starts at state σ_q^0. Upon receiving and processing a message m_0 from process r, q enters a new state σ_q^1. It then switches to state σ_q^2 as it processes the message m_1 from p. Interval σ_r^1 is said to be dependent on both σ_r^0 and σ_q^2, whereas σ_q^2, in turn, is dependent on both σ_q^1 and σ_p^0.

Each process periodically saves information about its state in a stable storage. This action is called *checkpointing*. A process may restart from the state contained in its last checkpoint after it crashes. However, messages sent by it to other processes, after its last checkpoint and before it crashes, may have already been processed by the receiving processes. Unless the recovered process is able to maintain the original dependency relation, the system may enter an inconsistent state. For the example shown in figure 1, assume process q crashes at time t and restarts from the latest checkpoint. If it first processes message m_1, instead of m_0, its successive states may be different from the original ones. In this case, state σ_q^2, which σ_r^1 depends on, no longer exists, and the system enters an inconsistent state.

In this paper, processors are assumed to be fail-stop [11] and communication channels are assumed to be reliable. In addition, messages sent from a source to a destination are first-in-first-out (FIFO) by using some end-to-end transmission protocols [16]. In our model, a message received by a node may not be immediately delivered to the addressed process for processing until it is requested.

III Recovery Structure

It is important that fault tolerance is provided in a transparent manner so that the design of applications can be simplified and straightforward. The structure of a node consists of an hierarchy of three layers, namely the application layer, the recovery layer, and the communication layer [12], as shown in figure 2. Application layer is solely responsible for the execu-

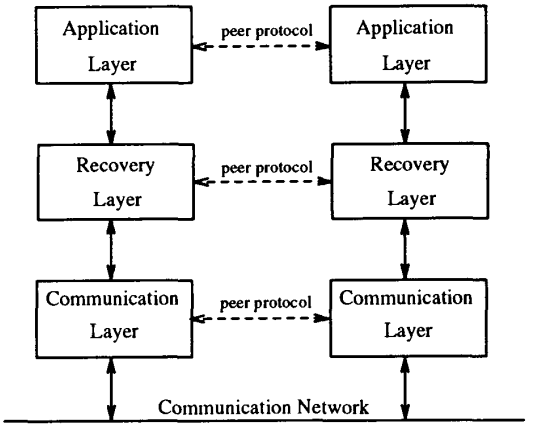

Figure 2: Three-layer system architecture.

tion of application processes. Processes communicate with each other by passing messages through recovery layer. Recovery layer provides fault tolerance to application layer in a transparent manner. Information required for processes to recover from crashes are maintained in this layer. It also takes necessary actions to restore a system to a consistent state after it crashes. Communication layer implements ene-to-end and broadcast communication using various protocols. All received messages are delivered to application layer via recovery layer. In the following, the recovery

layer is addressed in details.

III.1 Fundamental Concept of Recovery Layer

Inconsistency occurs if a process, after recovering from a crash, does not proceed in a way that maintains the dependency relation observed by other processes which are causally dependent on it. In effect, the order in which messages are processed by a process must be kept so that the history can be replayed when it recovers from a crash. This prevents other processes from rolling back.

The fundamental concept of our recovery mechanism is to maintain message processing order of each process in a given set of other processes. In case the process crashes, it restarts from its latest checkpoint and recovers to a consistent state by proceeding according to the information obtained from any of the surviving processes which keeps processing order for it. This scheme also avoids time-consuming operations of writing the aforementioned processing order into stable storage. Note that recovery layer periodically takes checkpoints for each process.

Now, consider a distributed process which consists of n individual application processes. Each application process is identified by a unique *id*. To facilitate the following discussion, let the n application processes form a single logical ring of length n. Strategies for forming various logical rings are discussed later in section V. When an application process sends a message to another process, it immediately circulates a short packet around the logical ring. The packet, denoted as *order message* (or *om*), contains a simple information about the order of messages it has already processed since the last time it sent a message. The recovery layer of each process maintains a table called *order table*, which contains a set of n lists of process *ids*. As a process receives an *om* from its predecessor on the ring, it updates the list in its order table, which corresponds to the originating process of *om*, with information contained in *om*. For example, figure 3(a) shows an execution of four distributed processes, p, q, r and s. The logical ring formed of them is illustrated in figure 3(b). Each process has a corresponding order table in the recovery layer. Each table consists of four lists, one for each of the processes. Order tables are initially empty. When process q sends m_1 to process s, it also immediately circulates an *om* on the ring. In this case, the *om* first reaches r, then s and so on. The content of this *om* is a pair of process *ids*, (q, p). The first attribute represents the originator of *om*, which is q in this case. The second attribute indicates

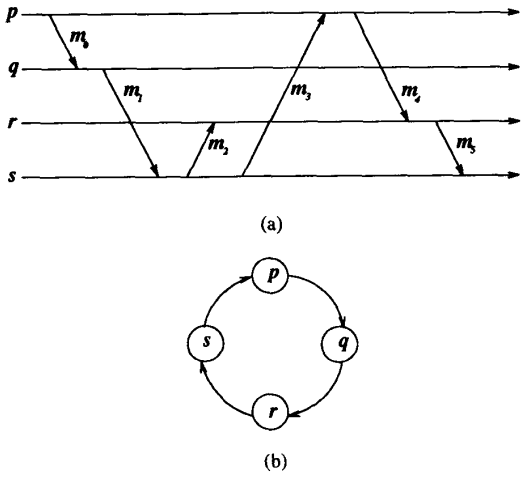

Figure 3: (a) A distributed computation with four processes. (b) The logical ring.

that the state transition interval of q, in which m_1 is sent, is initiated as a result of processing a message, m_0, which was previously sent from process p to q. As process r receives *om*, it updates the list in its order table, which corresponds to q, by appending the *id* p at the end of it. As a result, the first entry of the list is p. The other processes on this ring would act the same upon receiving *om*. The order table of r is illustrated in figure 4. Essentially, order table keeps processing order so that dependency relation among processes can be maintained. Similarly, s circulates

process id / order list	p	q	r	s
		p	s	q
			p	

Figure 4: Order table for process r shown in figure 3.

an *om*, (s, q), around the ring after it sends m_2 to r, but no circulation of *om* is needed after it sends m_3. Consider message m_5. Process r has not sent out any message in the state transition interval initiated by processing m_2. Consequently, as m_5 is sent to process s, the associated *om* must include two *id* pairs -- (r, s) and (r, p). In other words, an *om* should contain all the information about the processing order since its last transmission. Each *om* eventually returns to its originator and is thus consumed by the recovery layer.

Note that an order message originating from a process contains information about all incoming messages that have already been processed by it since last transmission. Hence, an *om* carries complete order information in-between two transmissions. Order messages from a process arrive at each node in the order of their circulations. The above observation shows that each list in an order table correctly record the message processing sequence of the corresponding process during fail-free operation.

III.2 Components of Recovery Layer

Recovery layer acts as an intermediate device between application and communication layers. It buffers data messages to and from application processes. Furthermore, it handles order messages circulating on the logical ring.

Figure 5 illustrates the internal structure of recovery layer. For each process, there is a set of input

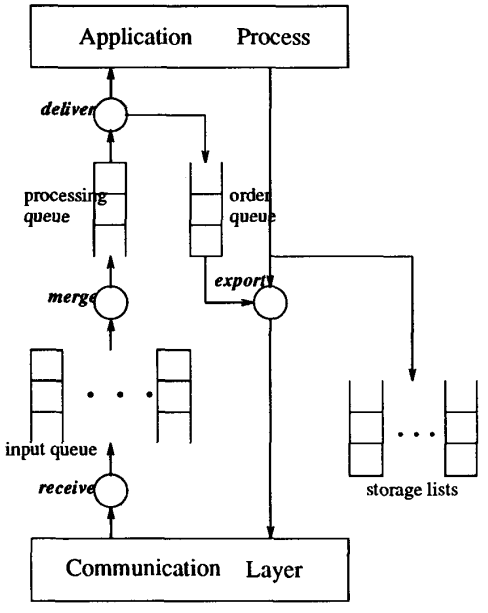

Figure 5: Internal structure of recovery layer.

queues, each corresponding to another process. Incoming data messages from other processes are stored in the corresponding input queues. The merge function combines various input queues, and places merged data into a processing queue [15]. This merge function reflects the property of nondeterminism in message processing with which our distributed system operates. These queues are all first-in-first-out (FIFO). A process makes a request for an incoming message to recovery layer. The recovery layer responds with delivering the data message at the head of the processing queue to the process. In the meantime, a pair of process *ids* is generated as described earlier, and placed in a queue called *order queue*. Therefore, the order of the messages inside the processing queue represents the sequence with which application process processes incoming messages. A set of storage lists, one corresponding to each other process, is used to store data messages sent to other processes. These storage lists are maintained for the purpose of retransmissions in recovery procedure [6, 14]. Whenever a process sends a message, the *export* function is called upon to pass the data message onto the communication layer for actual transmission. In the meantime, it also encapsulates all *id* pairs in the order queue into a packet, and circulates the packet around the logical ring immediately.

IV Algorithms and Protocols in Recovery Layer

In this section a set of algorithms and protocols used in the recovery layer is presented. If crashes occur in a system, a recovery procedure is followed to bring the system back to a consistent state. We show that the recovery mechanism presented below works correctly as long as order information is not totally lost in the system.

(a) Description of Algorithms

In the following, p denotes the current process and q denotes opposite process during message transmission. A procedure *deliver* is used to deliver valid data messages to applications.

Procedure *deliver*
begin
 if request for an incoming message;
  begin
   return message at head of processing queue;
   generate (p, *sou_id*), append it to order queue;
   $Tot\_Deliver_p[sou\_id]$++;
  end
end

As an application process makes a request for an incoming message, *deliver* function fetches the message at the head of processing queue and delivers it to the application for processing. In the meantime, a process-*id* pair is stored in the order queue. *sou_id* identifies the source process of data message. The variable $Tot\_Deliver_p[sou\_id]$ is used to indicate the total

number of messages sent from process *sou_id*, that has already been processed by p.

The *export* procedure shown below is used to handle the output of data messages from p to others.

```
Procedure export
begin
 Tot_Send_p[q]++;
 if (Tot_Send_p[q] ≤ Msg_Saved_q[p]);
   discard the application message;
 else
   begin
    place data message in corresponding storage
     list;
    if (Tot_Send_p[q] > Tot_Receive_q[p])
      begin
       pass data message to communication layer;
       if (order queue not empty)
         circulate an om with an incarnation
          number and append it to order list;
      end
   end
end
```

Normally, *export* function simply passes output data message onto communication layer for transmission. It then immediately circulates an *om* around the ring. An incarnation number, which indicates the number of checkpoints current process has taken, is added to the *om* to identify whether it is out-of-date so that an order table can exhibit correct processing sequence. Order queue of the current process is emptied after each message transmission. The *export* function also sees that data messages sent are not duplicated. This can be done by examing the values of two variables, $Msg\_Saved_q[p]$ and $Tot\_Receive_q[p]$. $Msg\_Saved_q[p]$ represents the number of messages sent by p to q which have already been processed by q before it's latest checkpoint. The variable $Tot\_Receive_q[p]$ contains the total number of messages sent from p that have been received by q. Both $Msg\_Saved_q[p]$ and $Tot\_Receive_q[p]$ are provided by q, for the purpose of garbage collection, when p broadcasts its recovery request after it crashes.

Upon taking a checkpoint, recovery layer broadcasts a checkpoint message so that other processes can perform related garbage collection. Two arguments are included in a checkpoint message. They are the incarnation number of the checkpointing process p and a vector $Msg\_Saved_p[q]$ $\forall q \neq p$. Garbage collection task is then carried out for the current process p; both the list in order table which corresponds to itself and its order queue are cleaned.

```
Procedure checkpoint
begin
 Msg_Saved_p[q] ⟵ Tot_Deliver_p[q]   ∀q ≠ p;
 save current state including Tot_Send_p[],
  Msg_Saved_q[], incar_num_p, and storage lists
  to stable storage;
 incar_num_p++;
 broadcast checkpoint message(incar_num_p,
  Msg_Saved_p[]);
 clear order list and order queue;
end
```

The messages received by a recovery layer can be categorized into two classes– data messages from other processes and recovery-related control messages. Different actions must be taken in response to receipts of various messages. The *receive* function shown below illustrates these actions.

```
Procedure receive
begin
 m ⟵ message from communication layer;
 switch (m)
 case (data message)
   if (in recovery state)
     if (q's message block already received)
       begin
        put message in input queue;
        Tot_Receive_p[q]++;
       end
     else
       discard the message;
   else
     begin
      put message in input queue;
      Tot_Receive_p[q]++;
     end
     put message in input queue;
   exit;
 case (om)
   begin
    if (in recovery state)
      if (order list corresponding to originator
        of om has been received)
        if (om.incar_num <
             INCAR_NUM[originator])
          discard the om;
        else
          append om to order list, and
           circulate om to next node;
      else
        discard the om;
```

```
    else
      if (originator of om is myself)
        consume the om;
      else
        if (om.incar_num <
            INCAR_NUM[originator])
          discard the om;
        else
          append om to order list, and
            circulate om to next node;
  end
  exit;
case (message block)
  put messages in input queue and increment
    Tot_Receive_p[q] by number of messages in
    the block;
  exit;
case (need_help) {ask for lost order list}
  begin
    if (in recovery state)
      circulate need_help to next node;
    else
      circulate requested order list, and
        consume need_help message;
  end
  exit;
case (ol) {order list for recovery}
  begin
    if (sender of ol is myself)
      consume the ol;
    else
      use ol to reconstruct corresponding order
        list, and circulate ol to next node;
  end
  exit;
case (Tot_Receive_q[p], Msg_Saved_q[p], or
      incar_num_q)
  update Tot_Receive_q[p], Msg_Saved_q[p], or
    INCAR_NUM_p[q];
  exit;
case (checkpoint message(incar_num_q,
      Msg_Saved_q[]))
  begin
    remove oms in order list corresponding to q,
      in which om.incar_num < incar_num_q;
    remove all messages before Msg_Saved_q[p] in
      storage list or stable storage for q;
    update Msg_Saved_q[p];
  end
  exit;
case (recovery message(Msg_Saved_q[p]))
  begin
    if (in normal state)
      circulate order list corresponding to myself;
    send message block containing data messages
      behind Msg_Saved_q[p] to q;
    send Tot_Receive_p[q], Msg_Saved_p[q] and
      incar_num_p to q;
  end
  exit;
end
```

Another algorithm called *recovery* guarantees consistent recovery of a distributed system.

```
Procedure recovery
begin
  get Tot_Send_p[] and Msg_Saved_p[] from stable
    storage;
  Tot_Receive_p[] ⟵ Msg_Saved_p[];
  Tot_Deliver_p[] ⟵ Msg_Saved_p[];
  disable generation of id pairs and merge function;
  broadcast recovery message(Msg_Saved_p[]);
  circulate a need_help message, in which myself
    is identified;
  wait for order lists, Tot_Receive_q[p],
    Msg_Saved_q[p] and incar_num_q ∀q ≠ p;
  merge messages in input queues according to order
    specified in order list corresponding to p;
  restore its state to the latest checkpoint;
  restart application process;
  while (incoming messages as per order list are not
         consumed)
    do nothing;
  enable generation of id pairs and merge function;
end
```

In a recovery state, a process must broadcast its demand to others. Normal generation of order messages must be disabled to avoid circulating unnecessary order information. The *merge* function is stopped to allow recovery layer to enforce previously defined message processing order.

(b) **Correctness**

A correct recovery in a distributed system relies on informing crashed processes of their message processing order prior to their failures so that history can be replayed. Consider some processes crash in the system. Assume that, for each crashed process, there is at least one surviving process which maintains processing order information after its latest checkpoint

for it. Consider a failed process p. Let σ_p^m be the last state transition interval of p in which p sent a message to another process, i.e. σ_p^m is the last interval having effect on other processes. In the following, we show that the proposed mechanism successfully recovers the system.

Lemma 1 *For any process r, the corresponding list in the order table of process $r1$ must be at least as up-to-date as that of process $r2$ if $r1$ precedes $r2$ in the ring with respect to r.*

Proof: If an order message has reached $r2$, it must have already passed through $r1$. Hence, its corresponding information must have been reflected in the order table of $r1$. □

Lemma 2 *Every process, regardless of its state, will obtain the up-to-date list of order table, which corresponds to p, during recovering procedure.*

Proof: Assume l is the first fail-free process found by traversing the ring from p. From lemma 1 and the condition specified earlier, we know that l must contain the most up-to-date order information regarding p. According to the *receive* function and the associated recovery procedure, l will circulate this list around the ring and will eventually reach every process. Hence, every process will have a consistent view of the list which corresponds to p. This up-to-date information contains all the order information up to σ_p^m. Since the corresponding list in every order table on the ring is updated with the newly circulated one and the upcoming *om* originating from p is generated after execution is resumed, the list is guaranteed to be correct eventually. □

Since an *om* originating from a fail-free process may be lost due to failure of another process, circulation of an order list by a fail-free process around the ring makes sure that every process can correctly reconstruct its corresponding list in the order table. The following lemma summarizes the statement.

Lemma 3 *Every process, regardless of their states, will obtain the up-to-date order table during recovering procedure.*

A crashed process requires lost data messages, in addition to its processing order information, to recover to a consistenet state.

Lemma 4 *For process p, all data messages from its latest checkpoint upto the point of its crash can be obtained eventually.*

Proof: There are two cases to be considered.
Case 1: the sender of a data message is fail-free:
The data message exists in recovery layer of the sender. According to our mechanism, this data message will be resent to p.
Case 2: the sender, say l, of the data message crashes:
If the data message is sent before l took its latest checkpoint, the message already exists in l's stable storage, and hence can be resent. Consider that l has to resume its execution to generate this data message. Assume S_f denotes the set of crashed processes. Let mg_i represent the first message required by a failed process i, $i \in S_f$, which must be generated by re-execution of another process in S_f. Let the source of mg_p be l. The state transition interval of l, in which mg_p is sent, can not happen before that initiated by mg_l. Consequently, there can not exist a cycle among a set of crashed processes with respect to this type of messages. As the number of processes is limited, there must exist a process which can initiate the execution, and eventually p will obtain the required data message mg_p for recovering. By the same token, other data messages of this type, which are required by p, can be obtained as well. □

Theorem 1 *The recovery mechanism can successfully recover a distributed process to a consistent state after the system crashes. In addition, only the failed processes need to roll back to their latest checkpoints.*

Proof: From lemma 3, a failed process has complete information about its previous processing order. In addition, it can obtain required data messages from other processes, regardless of failing or not, by lemma 4. A failed process can resume its execution using these informations along with its latest checkpoint. No surviving process has to roll back. □

V Fault-Tolerant Resiliency

In order to reduce overhead due to circulation of order message, one can construct a logical ring of smaller size. The aforementioned mechanisms would work with only minor modification. In fact, the sizes of logical rings for application processes can be varying. Depending on the criticality of an application process, the logical ring can be tailored to fit its individual needs.

Selection of logical ring can affect fault-tolerant capability provided by a system. As described earlier we do not allow information about message processing order be totally lost as we do not record order information in stable storage. To attain this goal, one can tailor the logical rings to make a system reliable.

For example, a process should place more reliable nodes at the beginning part of its logical ring so as to reduce the probability of losing its order messages due to failures. Apparently, there is a tradeoff between the length of a ring, which reflects the resiliency of fault tolerance, and the amount of overhead due to *om* traffic. A longer ring tends to provide better fault tolerance while incurs more overheads.

VI Conclusions

In this paper, a novel approach is proposed for implementing checkpoint/rollback-recovery in distributed computing systems. It takes advantages of a logical-ring structure, in which message passing is sequential, to facilitate the maintenence of information about message processing order of each process. As a result, the associated fail-free overhead is moderate in contrast to previous methods. When failures occur, only the failed processes need to roll back to their latest checkpoints, whereas surviving processes can continue execution without being blocked. Recovery mechanism is simple and works as long as order information is not totally lost due to simultaneous failures in the system. In the future, we are intending to further exploit the property of ring-based crash recovery methods. We are currently investigating the possibility of combining the property of the proposed method with that of others that append additional information to a data message [4, 8].

References

[1] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under unix. *ACM Trans. Comput. Syst.*, 7(1):1–24, February 1989.

[2] K. M. Chandy and L. Lamport. Distributed snapshot:determining global states of distributed system. *ACM Trans. on Computer systems*, 3:63–75, 1985.

[3] F. Cristian and F. Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proc. 10th Symp. Reliable Distributed Systems*, pages 12–20, September 1991.

[4] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Computers*, 41(5):526–531, May 1992.

[5] D. B. John and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *J. Algorithms*, 11(3):462–491, September 1990.

[6] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Proc. Conf. on Fault-Tolerant Computing Systems*, pages 14–19, 1987.

[7] T. T-Y. Juang and S. Venkatesan. Efficient algorithms for crash recovery in distributed systems. In *10th Conf. on Foundations on Software Technology and Theoretical Computer Science*, pages 349–361, 1990.

[8] T. T-Y. Juang and S. Venkatesan. Crash recovery with little overhead. In *Int. Conf. on Distributed Computing Systems*, pages 454–461, 1991.

[9] R. Koo and S. Toueg. Checkpoint and rollback-recovery for distributed systems. *IEEE Trans. Software Engineering*, SE-13(1):23–31, January 1987.

[10] M. L. Powell and D. L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proc. 9th ACM Symp. Operat. Syst. Principles*, pages 100–109, October 1983.

[11] R. D. Schlichting and F. B. Schneider. Fail-stop processors. *ACM Trans. Comput. Syst.*, 1(3):222–238, August 1983.

[12] S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs, S. Tao, and A. Tully. Principal features of the voltan family of reliable node architectures for distributed systems. *IEEE Trans. Computers*, 41(5):542–549, May 1992.

[13] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *ACM Symp. Principles Distributed Compt.*, pages 223–238, August 1989.

[14] R. E. Strom, D. F. Bacon, and S. A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *Proc. Conf. on Fault-Tolerant Computing Systems*, pages 44–49, 1988.

[15] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. on Computer Systems*, 3(3):204–226, August 1985.

[16] A. S. Tannenbaum. *Computer Networks*. Prentice Hall, Inc., 1981.