Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit *

Elmootazbellah N. Elnozahy and Willy Zwaenepoel

Abstract

Manetho is a new transparent rollback-recovery protocol for long-running distributed computations. It uses a novel combination of *antecedence graph* maintenance, uncoordinated checkpointing, and sender-based message logging. Manetho simultaneously achieves the advantages of pessimistic message logging, namely limited rollback and fast output commit, and the advantage of optimistic message logging, namely low failure-free overhead. These advantages come at the expense of a complex recovery scheme.

Index Terms: Antecedence graph, checkpointing, message logging, rollback-recovery, transparent fault tolerance.

1 Introduction

Transparent rollback-recovery is an attractive approach for providing fault tolerance to long-running distributed applications without real-time requirements. Three key performance

*This research was supported by NSF Grants CDA-8619893 and CCR-9116343, by IBM Corporation under Research Agreement No. 20170041, and by an IBM Graduate Fellowship. The authors are with the Department of Computer Science, Rice University, Houston, TX 77251-1892.

considerations in this approach are failure-free overhead, extent of rollback, and output commit latency. During failure-free operation, a rollback-recovery protocol records information about the computation's execution on stable storage, thereby causing failure-free overhead. The system uses this information after a failure to roll the computation back to a consistent state [2]. The system also invokes an output commit algorithm each time the computation sends a message to the "outside world." The outside world consists of the entities that cannot roll their states back (a line printer, for instance). Because the output commit algorithm must ensure that the state from which the message is sent will never be rolled back [16], it may introduce latency in sending messages to the outside world.

Existing transparent rollback-recovery methods fall into three classes: pessimistic message logging, optimistic message logging, and consistent checkpointing. These methods achieve only a subset of the goals of reducing the overhead during failure-free operation, limiting the extent of rollback, and reducing the latency of output commit. Pessimistic message logging protocols limit rollback by synchronously logging recovery information on stable storage [1, 12]. A failed process is restored to its state before the failure, and processes that survive the failure are not rolled back. In addition, no latency is incurred in sending messages to the outside world. Synchronous logging of recovery information however results in high failure-free overhead, unless special-purpose hardware is used. Optimistic message logging protocols [6, 7, 15, 16] reduce failure-free overhead by logging recovery information asynchronously. Processes that survive a failure may however be rolled back. Furthermore, the latency of output commit is higher than in pessimistic message logging since a message cannot be sent to the outside world without multi-host coordination. Consistent checkpointing protocols [2, 4, 8, 10, 17, 18] do not cause failure-free overhead, except while a consistent checkpoint is being taken. Processes that survive a failure may however be rolled back, and output commit may require taking a multi-host consistent checkpoint, resulting in considerable latency.

Manetho is a new transparent rollback-recovery protocol that, unlike existing protocols, simultaneously achieves the goals of low overhead, limited rollback, and fast output commit. It achieves these goals by using an *antecedence graph*, which records the "happened before" [9] relationship between certain events in the computation, in combination with uncoordinated checkpointing and sender-based volatile message logging [5]. Manetho avoids synchronous logging of recovery information on stable storage most of the time, thereby reducing the overhead during failure-free operation. Manetho also reduces the latency of output commit by allowing messages to be sent to the outside world without multi-host coordination. After a failure, surviving processes are not rolled back, and failed processes are rolled back only to their most recent checkpoints. The protocol tolerates an arbitrary number of fail-stop failures [14], including failures during recovery, and avoids the domino effect [13].

Manetho's advantages come at the expense of a complex recovery scheme and some limitations on support for nondeterminism. The expense of recovery should not be a major concern in modern systems where failures are expected to be infrequent. Nondeterminism is limited to message receipt or other nondeterministic events that can be efficiently recorded in the antecedence graph and replayed during recovery.

Experience with an implementation of Manetho shows that the overhead of maintaining the antecedence graph and message logs is small [3]. We concentrate in this paper on the protocol description and its correctness. Implementation, performance, and scalability are considered elsewhere [3].

2 Assumptions

We assume that the computation consists of a number of fail-stop [14] *recovery units* (RUs) [16] which communicate only by messages over an asynchronous network. An RU consists of one or more threads that manipulate the RU's internal state. Each RU has access to a stable storage device. A failed RU can be restarted on *any* available machine.

The execution of an RU consists of a sequence of piecewise deterministic state intervals [16], each started by a nondeterministic event. Such an event can be 1) the receipt of a message, 2) an internal nondeterministic event such as a kernel call or a synchronization operation between two threads within the same RU, or 3) the creation of the RU.

Figure 1 shows the execution of three RUs and their state intervals. A horizontal line represents the execution of each RU. An arrow between two horizontal lines denotes a message, and a vertical bar marks the beginning of each state interval. The notation σ_i^p denotes the i^{th} state interval of RU p, where i is referred to as the *index* of σ_i^p. Each application message has a system-wide unique identifier.

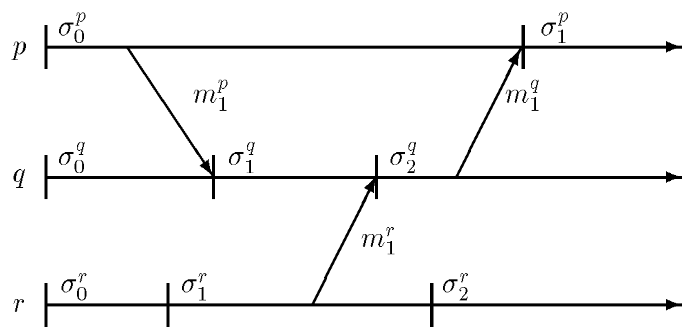

Figure 1 An Example Execution.

We do *not* assume that the communication network is reliable: messages may be lost, duplicated, delivered out of order, or arbitrarily delayed. However, we assume that the network is immune to partition.

Input received by an RU from the outside world must be saved on stable storage before that RU can send a message to another RU or to the outside world, because the outside world cannot be relied upon to replay the input during recovery.

3 The Antecedence Graph

The antecedence graph (AG) of a state interval σ_i^p, $AG(\sigma_i^p)$, is a directed acyclic graph. It contains a node representing σ_i^p and a node for each state interval that "happened before" [9] σ_i^p. Figure 2 shows $AG(\sigma_1^p)$ corresponding to the example of Figure 1.

For a state interval created by the receipt of a message, the corresponding AG node has two incoming edges: one from the node representing the previous state interval in the receiving RU and one from the node representing the state interval from which the message was sent. The node contains: 1) a type field that indicates a message receipt, 2) the identifier of the receiver, 3) the identifier of the sender, 4) the index of the created state interval, and 5) the unique identifier of the message. The AG does *not* contain a copy of the message's data.

For a state interval created by an internal nondeterministic event, the corresponding AG node has one incoming edge from the node representing the previous state interval of the same RU. Such a node contains a field that indicates the type of the event and the information necessary to replay the event during recovery.

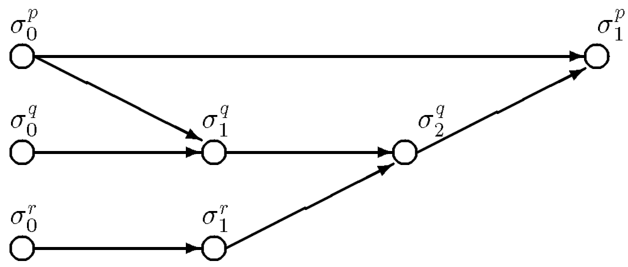

Figure 2 Antecedence Graph of state interval σ_1^p, $AG(\sigma_1^p)$.

4 Failure-Free Operation

4.1 Information in Volatile Storage

Each RU maintains in volatile memory the AG of its current state interval, and a log that contains the data and identifier of each message it *sends*. When an RU sends a message, it (conceptually) piggybacks the AG of its current state interval on the message. The receipt of the message starts a new state interval in the receiving RU, and the AG of that state interval is constructed from the AG of the previous state interval and the AG piggybacked on the message, as described in Section 3.

The sender need not include the complete AG of its current state interval in each message. Instead, incremental piggybacking is used. By definition, $AG(\sigma_i^p)$ is a proper subgraph of $AG(\sigma_{i+1}^p)$. Each RU q that communicates with p includes with each message sent to p the maximum state interval index j such that the node representing σ_j^p is in q's AG. Later, when p sends a message to q from some σ_i^p, it appends only $AG(\sigma_i^p) - AG(\sigma_j^p)$.

4.2 Information on Stable Storage

Periodically, each RU records a checkpoint of its state on stable storage. The checkpoint is not coordinated with the other RUs in the computation. While recording the checkpoint, the RU also saves the volatile message log and the AG of its current state interval on stable storage.

Occasionally, each RU asynchronously saves the AG of its current state interval on stable storage. The subgraph on stable storage need not be piggybacked on outgoing messages, avoiding the need to piggyback large AGs. An AG at some RU may be missing one or more subgraphs, but these missing subgraphs are always available on stable storage.

Before sending a message to the outside world, an RU saves the AG of its current state interval on stable storage (output commit). No coordination with other RUs is necessary.

4.3 Incarnation Numbers

Because of network delay, a message m_i^p that originates from RU p may arrive at its destination q after p has failed. If q has been notified of p's failure before receiving m_i^p, q will not be able to determine whether m_i^p originated before or after p's failure. This is an instance of the problem of ordering the perception of failures with respect to messages.

To solve this problem, each RU starts a new incarnation [16] at the beginning of each recovery. Each incarnation is identified by a monotonically increasing incarnation number, and each message is tagged with the current incarnation number of the sender. When an RU starts the recovery protocol, it reliably informs the other RUs in the computation of its new incarnation number before proceeding. Messages tagged with old incarnation numbers are rejected.

5 Recovery Protocol

5.1 Overview

Figure 3 shows the recovery protocol. A recovering RU p restores its state, message log, incarnation number, and AG from stable storage. Then, RU p calls the procedure *RECOVER*, passing as arguments the RU's identifier p, the state interval index of the checkpointed state c, the incarnation number $INCNUM$, and the set S containing the RUs that participate in the computation.

RU p increments its incarnation number $INCNUM$ and saves it on stable storage. The graph G is initialized to the AG that was retrieved from stable storage, $AG(\sigma_c^p)$. RU p then calls the procedure *GET_AG* at each other RU. Messages exchanged for the purpose of recovery are out-of-band and do not carry AG information. An end-to-end communication protocol is used to ensure reliable remote procedure call delivery.

In *GET_AG* at RU q, q first saves the AG of its current state interval on stable storage. RU q determines σ_k^p, the most recent state interval of p that has a node in q's AG. Next, q adds k to $REJECTVEC$, and until q receives a *SEND_INC* call from p (see below), q rejects any *application* message (from any sender) whose piggybacked AG contains a node for any state interval σ_i^p, where $i > k$. Then, q returns its incarnation number and $AG(\sigma_k^p)$. Note that recovering RUs respond to *GET_AG* calls.

When a *GET_AG* call returns, p merges the returned AG into G, and includes q's incarnation number in the incarnation number vector $INCVEC$. After all *GET_AG* calls have

```
procedure RECOVER(p, c, INCNUM, S)
    INCNUM ← INCNUM + 1;
    save INCNUM on stable storage;
    INCVEC[p] ← INCNUM;
    G ← AG(σ_c^p);

    for all q ∈ S, q ≠ p do
        (INQ, AGQ) ← remote call at q : GET_AG(p);
        G ← G ∪ AGQ;
        INCVEC[q] ← INQ;
    for all q ∈ S, q ≠ p do
        remote call at q : SEND_INC(p, INCVEC);

    m ← max j such that σ_j^p ∈ G;
    STATEINDEX ← c;
    while STATEINDEX ≤ m do
        execute up to next event without sending
            application messages;
        STATEINDEX ← STATEINDEX +1;
        if next event is a receive then
            request message from sender's log;
        else
            re-execute internal event;
    return;


procedure GET_AG(p)
    save AG on stable storage;
    k ← max j such that σ_j^p ∈ AG;
    REJECTVEC[p] ← k;
    return (INCNUM, AG(σ_k^p));


procedure SEND_INC (p, PINCVEC)
    for all s ∈ S do
        INCVEC[s] ← max(INCVEC[s], PINCVEC[s]);
    REJECTVEC[p] ← ∞;
    return;
```

Figure 3 The Recovery Protocol.

returned, p calls the procedure *SEND_INC* at every other RU, with p and *INCVEC* as arguments. In *SEND_INC* at RU q, q updates its incarnation number vector and removes the restrictions on accepting messages that contain state intervals of p.

RU p proceeds to recreate the pre-failure execution up to state interval σ_m^p. During recovery, p requests messages from their senders' logs and re-executes internal events, as necessary. RU p does not send messages to the outside world or to other RUs, but it stores the messages that it would have sent in its volatile message log.

5.2 Correctness

We first show that the graph G computed by *RECOVER* is indeed $AG(\sigma_m^p)$.

Lemma 1 $G = AG(\sigma_m^p)$.

Proof We show that $G \subseteq AG(\sigma_m^p)$ and $AG(\sigma_m^p) \subseteq G$.

$G \subseteq AG(\sigma_m^p)$: Since initially $G = AG(\sigma_c^p)$, then $\forall g \in G$, $g \in AG(\sigma_c^p) \vee g \in G - AG(\sigma_c^p)$.

Case 1: $g \in AG(\sigma_c^p)$. Since $c \leq m$, $AG(\sigma_c^p) \subseteq AG(\sigma_m^p)$. Thus $g \in AG(\sigma_m^p)$.

Case 2: $g \in G - AG(\sigma_c^p)$. $\exists$ q such that in p's *GET_AG* call at q, $g \in AG(\sigma_k^p)$. Since $k \leq m$, $AG(\sigma_k^p) \subseteq AG(\sigma_m^p)$. Thus, $g \in AG(\sigma_m^p)$.

$AG(\sigma_m^p) \subseteq G$: If $c = m$ then obvious. If $c < m$, then let q be the RU that returned $AG(\sigma_m^p)$ to p's *GET_AG* call. If q has the complete graph $AG(\sigma_m^p)$ in its own AG, then $AG(\sigma_m^p) \subseteq G$. Otherwise, the returned $AG(\sigma_m^p)$ must be missing one or more subgraphs. This can happen only because some other RUs have saved the missing subgraphs on stable storage before sending the messages that should have included them. These RUs will return the missing subgraphs during p's *GET_AG* calls, regardless of any failure. $\square$

Definition 1 *σ_i^p is a* lost state interval *of RU p if and only if σ_i^p occurred during some incarnation v of RU p, and RECOVER at the beginning of incarnation $v+1$ restores p only up to some state interval σ_m^p, where $m < i$.*

Lemma 2 *After all GET_AG calls in p's recovery return, but before p sends any SEND_INC calls, no AG of any RU contains a node representing a lost state interval of p.*

Proof When p's *GET_AG* call executes at any *RU* q, no state interval σ_i^p, such that $i > m$, has a corresponding node in the *AG* of *RU* q. After returning p's *GET_AG* call and before receiving p's *SEND_INC* call, the use of *REJECTVEC* prevents *RU* q from accepting any message whose piggybacked *AG* carries a node that represents σ_i^p, where $i > m$. $\square$

Because of arbitrary delays, the network may contain a message whose piggybacked *AG* has a node that represents a lost state interval of *RU* p. We show that such a message will be rejected.

Lemma 3 *A message whose piggybacked AG contains a node that represents a lost state interval of p will be rejected by any RU that receives it.*

Proof Assume that *RU* r sends to *RU* q a message m_j^r whose piggybacked *AG* contains a node that represents a lost state interval σ_i^p, $i > m$. From Lemma 2, m_j^r cannot originate from the current incarnation of r. Hence, m_j^r must originate from a previous incarnation of r. There are three cases:

Case 1: m_j^r arrives at q before p's *GET_AG* call executes at q. This is impossible since q would have returned $AG(\sigma_i^p)$ during p's *GET_AG* call, with $i > m$, a contradiction.

Case 2: m_j^r arrives at q after p's *GET_AG* call executes at q, but before p's *SEND_INC* call executes at q. The message will be rejected because of the use of *REJECTVEC* as in Lemma 2.

Case 3: m_j^r arrives at q after p's *SEND_INC* call executes at q. Because p's *SEND_INC* call contains the current incarnation number of every RU, q detects that the incarnation of r tagging m_j^r is old and rejects it. $\square$

We next show that despite an arbitrary number of failures, including additional failures during recovery, RU p re-executes to the state interval σ_m^p.

Lemma 4 *$\forall\, i, q$ such that $\sigma_i^q \in G, AG(\sigma_i^q)$ will remain available at RU q.*

Proof If q is live when it returns p's *GET_AG* call, then the lemma is true regardless of any subsequent failures of q, since q saves its AG on stable storage during *GET_AG*. Otherwise, RU q was recovering when it returned p's *GET_AG* call. There are two cases:

Case 1: $AG(\sigma_i^q)$ is a subgraph of the AG of the current state interval of some live RU r that returned p's *GET_AG* call. There are three cases:

case i: r returned p's *GET_AG* call before q's *GET_AG* call executed at r. Thus, r has saved $AG(\sigma_i^q)$ on stable storage. Regardless of future failures of r or q, $AG(\sigma_i^q)$ will be returned to q during its *GET_AG* call at r.

case ii: r returned p's *GET_AG* call after q's *GET_AG* but before q's *SEND_INC*. Then $AG(\sigma_i^q)$ must have been returned to q's call, since r could not have added $AG(\sigma_i^q)$ to its own AG after q's call, from Lemmas 2 and 3. This is also true if r subsequently fails, because a recovering RU does not accept application messages until it finishes recovery.

case iii: r returned p's *GET_AG* call after q's *SEND_INC* call. Lemmas 2 and 3 show that σ_i^q cannot be a lost state, and therefore $AG(\sigma_i^q)$ is available at q.

Case 2: $AG(\sigma_i^q)$ is not a subgraph of the AG of the current state interval of any live RU. Hence, either $AG(\sigma_i^q) \subset AG(\sigma_c^p)$, in which case p returns $AG(\sigma_i^q)$ during q's *GET_AG*; or p must have received $AG(\sigma_i^q)$ from some RU r that was recovering and had $AG(\sigma_i^q)$ as a subgraph of its AG on stable storage, in which case both p and q will receive $AG(\sigma_i^q)$ from r, regardless of any subsequent failures of p, q or r. $\square$

Lemma 5 *The recovery protocol restores p up to state interval σ_m^p, despite any other failures in the system.*

Proof Construct graph F from G by removing the nodes that represent state intervals either in live RUs or that occurred prior to the most recent checkpoint of each recovering RU. Every state interval that has a corresponding node in F will be recreated, since the execution in each state interval is deterministic. The proof proceeds by induction on the topological sort of F, which must exist because F is acyclic.

Base case: Each node f at level 0 of the topological sort represents the first state interval after the checkpoint in a recovering RU. If f corresponds to an internal event, it contains the information necessary to recreate the state interval after restarting the execution from the checkpointed state. If f corresponds to a message receipt, then the source of the message must be the outside world, a state interval in a live RU, or a state interval that occurred before the checkpointed state in a recovering RU, from the construction of the graph F. In the first case, the message is available on stable storage. In the other two cases, the message is available in the sender's log and can be replayed.

Induction hypothesis: Assume that the lemma is true for all nodes at topological level k.

Induction step: For each node f at topological level $k+1$, if f corresponds to an internal event, then the corresponding state interval is recreated by starting execution from the previous state interval (which is reconstructed by the induction hypothesis) and using the information in f. If f corresponds to a message receipt, then the corresponding state interval is reconstructed by starting execution from the previous state interval and requesting the message to be replayed. The message is available either because it was recreated during recovery by the induction hypothesis, or because it was available in the log of a sender or on stable storage as in the base case. $\square$

Lemma 6 *The protocol is deadlock-free.*

Proof No deadlock can occur during the GET_AG calls, because recovering RUs return GET_AG calls. Lemma 5 also shows that no deadlock can occur during recreating the state intervals. $\square$

The next two lemmas establish limits on the amount of rollback during recovery.

Lemma 7 *Only one checkpoint for each RU needs be retained on stable storage.*

Proof Follows immediately from the construction in the proof of Lemma 5, since each RU restarts from its most recent checkpoint and recovers. $\square$

Lemma 8 *The recovery protocol avoids the domino effect.*

Proof A recovering RU rolls back only once and only to its last checkpoint, and no RU needs to roll back to replay the messages required for recovery of any other RU. $\square$

We next consider output commit. We show that all state intervals from which output is committed will be recovered, and that no output is committed from any lost state interval.

Lemma 9 *A state interval from which output is committed will be recovered.*

Proof Before committing output, an RU saves its AG on stable storage. The AG will be available despite any subsequent failure, and lemma 5 shows that the state interval can be recovered. $\square$

Lemma 10 *No output is committed from any lost state interval.*

Proof By contradiction. If output were committed from a lost state interval, then its AG would have been saved on stable storage, then the state interval could not be lost. $\square$

Definition 2 *Two distributed computations are equivalent if and only if both produce the same sequence of output.*

Theorem 1 *A failure-prone computation is equivalent to some failure-free computation that starts from the same initial state.*

Proof Let C be a failure-prone computation. Derive the failure-free computation C' from C by removing failures, recoveries, lost state intervals, and messages rejected either due to old incarnation numbers or due to the use of $REJECTVEC$. Lemmas 4, 5, 6 and 9 show that output sent in C will be sent in C'. Lemmas 2, 3, and 10 show that no output will be sent in C' that was not sent in C.

We now show that the state intervals constituting C' could happen in a failure-free execution that starts in the same initial state as C. The proof proceeds by induction on the state intervals of C'.

Base case: The initial state of each RU occurs in both C and C'.

Induction hypothesis: Assume that the subset of C' consisting of all state intervals that "happened before" σ_i^p can occur in a failure-free execution.

Induction step: We show that the subset of C' consisting of state interval σ_i^p and all state intervals that "happened before" σ_i^p can occur in a failure-free execution. Consider the state transition from state interval σ_{i-1}^p to σ_i^p in C'. The execution during σ_{i-1}^p is deterministic. Therefore, by replaying the event that created σ_i^p in C, the same transition from σ_{i-1}^p to σ_i^p occurs in C'. $\square$

6 Garbage Collection

To reclaim space from the message log, an RU p may decide that every message sent before some state interval σ_i^p is to be discarded. From Lemma 7, no RU will roll back beyond its latest checkpoint. Therefore, for each RU q such that p has sent a message to q before σ_i^p, p requests that q take a checkpoint if q has indeed received the message and has not taken a checkpoint since. In practice, σ_i^p is chosen such that forcing checkpoints is rarely necessary. After all RUs acknowledge its request, p can safely discard all messages sent before σ_i^p. Each RU maintains a list that contains the index of the state interval at each other RU before which all messages sent were garbage collected. This list is used to reject messages that arrive at their destinations after their copies in their senders' logs have been garbage collected. We discuss recovery of garbage collection information elsewhere [3].

To reclaim the space used by the AG, an RU can discard a node that corresponds to σ_i^q, if q has taken a checkpoint at state interval σ_c^q, where $c \geq i$. By Lemma 7, the information in that node will no longer be needed during recovery. For the purpose of garbage collection of AG, RUs occasionally exchange the state interval indexes of their most recent checkpoints.

7 Related Work

Several systems use message replay for rollback-recovery [1, 5, 6, 7, 11, 12, 15, 16]. Except for the Psync recovery protocol [11], none of these systems use a graph that records the "happened before" relation [9] between certain events. The combination of the antecedence graph with uncoordinated checkpointing and sender-based volatile message logging allows Manetho to achieve its goals of low failure-free overhead, limited rollback, and fast output commit, albeit at the expense of a more complex recovery protocol.

Manetho's antecedence graph differs from Psync's *context graph* [11] in that the antecedence graph records the *order of message receipt* within the same RU, while Psync does not. The order of message receipt is *exactly* the information required for message replay during recovery. The same information can be deduced from the context graph by applying a deterministic ordering filter. This filter delays the delivery of each application message until several subsequent application messages are received [11]. Moreover, unlike the antecedence graph, the context graph requires that each process receives and logs every message exchanged in the system. However, Psync's context graph is meant to support a variety of applications, while Manetho's antecedence graph is specifically designed for rollback-recovery.

8 Conclusion

Manetho is a new transparent rollback-recovery protocol for long-running distributed computations. It achieves the advantages of pessimistic protocols, namely limited rollback and fast output commit, and the advantage of optimistic protocols, namely low overhead during failure-free operation. Manetho uses a novel combination of antecedence graph maintenance, uncoordinated checkpointing and sender-based message logging. This reduces overhead by avoiding synchronous logging of recovery information on stable storage most of the time. The latency of output commit is reduced by avoiding multi-host coordination. Sending a message to the outside world requires only a synchronous write of the local antecedence graph on stable storage. The protocol tolerates an arbitrary number of fail-stop failures, including additional failures during recovery. After a failure, surviving processes do not roll back, and failed processes roll back only to their most recent checkpoints. These advantages come at the expense of a complex recovery scheme and some limitations on nondeterminism.

References

[1] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[2] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[3] E.N. Elnozahy and W. Zwaenepoel. Manetho: A low overhead rollback-recovery system with fast output commit. Technical Report TR91-152, Rice University, March 1991.

[4] F. Jahanian and F. Cristian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 12–20, Bologna, Italy, September 1991.

[5] D.B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pages 14–19, June 1987.

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

[7] T. Juang and S. Venkatesan. Crash recovery with little overhead. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 454–461, May 1991.

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

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] K. Li, J.F. Naughton, and J.S. Plank. Checkpointing multicomputer applications. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 1–10, October 1991.

[11] L.L. Peterson, N.C. Bucholz, and R.D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[12] M.L. Powell and D.L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 100–109, October 1983.

[13] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[14] R.D. Schlichting and F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[15] A.P. Sistla and J.L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, August 1989.

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

[17] K.-L. Wu and W.K. Fuchs. Recoverable distributed shared memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

[18] K.-L Wu, W.K. Fuchs, and J.H. Patel. Error recovery in shared memory multiprocessors using private caches. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):231–240, April 1990.