Efficient Transparent Optimistic Rollback Recovery for Distributed Application Programs

David B. Johnson

March 1993
CMU-CS-93-127

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A version of this paper will appear in
Proceedings of the 12th Symposium on Reliable Distributed Systems,
IEEE Computer Society, October 1993.

This research was supported in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035.

Keywords: Distributed systems, fault tolerance, rollback recovery, optimistic message logging, checkpointing, output commit.

Abstract

Existing rollback-recovery methods using consistent checkpointing may cause high overhead for applications that frequently send output to the "outside world," since a new consistent checkpoint must be written before the output can be committed, whereas existing methods using optimistic message logging may cause large delays in committing output, since processes may buffer received messages arbitrarily long before logging and may also delay propagating knowledge of their logging or checkpointing progress to other processes. This paper describes a new transparent rollback-recovery method that adds very little overhead to distributed application programs and efficiently supports the quick commit of all output to the outside world. Each process can independently choose at any time either to use checkpointing alone (as in consistent checkpointing) or to use optimistic message logging. The system is based on a new commit algorithm that requires communication with and information about the minimum number of other processes in the system, and supports the recovery of both deterministic and nondeterministic processes.

1. Introduction

In a distributed system, long-running distributed application programs run the risk of failing completely if even one of the processors on which they are executing should fail. For programs without real-time requirements, the use of rollback recovery can be an efficient method of providing fault tolerance that can recover the execution of the program after such a failure. During failure-free execution, state information about each process is saved on stable storage so that the process can later be restored to certain earlier states of its execution. After a failure, if a *consistent* system state can be formed from these individual process states, the program as a whole can be recovered and the resulting execution of the system will then be equivalent to some possible failure-free execution [Chandy 85]. Rollback recovery can be made completely *transparent* to the application program, avoiding special effort by the programmer and allowing existing programs to easily be made fault tolerant.

With any transparent rollback-recovery method, care must be taken in committing output to the *outside world*. The outside world consists of everything with which processes can communicate that does not participate in the system's rollback-recovery protocols, such as the user's workstation display, or even the file system if no special support is available for rolling back the contents of files. Messages sent to the outside world must be delayed until the system can guarantee, based on the state information that has been recorded on stable storage, that the message will never be "unsent" as a result of processes rolling back to recover from any possible future failure. If the sender is forced to roll back to a state before the message was sent, recovery of a consistent system state may be impossible, since the outside world cannot in general be rolled back. Once the system can meet this guarantee, the message may be *committed* by releasing it to the outside world.

Rollback-recovery methods using *consistent checkpointing* [Chandy 85, Cristian 91, Elnozahy 92a, Israel 89, Koo 87, Lai 87, Leu 88, Li 91, Moura e Silva 92, Ramanathan 88, Spezialetti 86, Tong 89, Venkatesh 87] record the states of each process separately on stable storage as a process checkpoint, and coordinate the checkpointing of the processes such that a consistent system state is always recorded by the collection of process checkpoints. The checkpoint of a process records the address space of the process, together with any kernel or server state information necessary to recreate the process executing in that state. To recover from any failure, each process can simply be restarted on any available processor from the state recorded in its most recent checkpoint.

However, before committing output to the outside world from a system using consistent checkpointing, a new consistent checkpoint must be recorded after the state from which the output was sent. This may involve each process writing large amounts of data to a stable storage server over the network. Even if sophisticated checkpointing methods are used that allow the checkpoint data to be written in parallel with the execution of the process [Elnozahy 92a, Johnson 89, Li 90], the output commit must wait until all of the data has been completely written to stable storage, significantly delaying the output. If new output messages are sent frequently to the outside world, the overhead caused by the necessary frequent checkpointing can severely degrade the performance of the system. In large systems, performance may be further degraded, as all processes write their checkpoints at the same time and the stable storage service or network could easily become a performance bottleneck.

On the other hand, rollback-recovery methods using *optimistic message logging* [Johnson 89, Johnson 90, Juang 91, Lowry 91, Sistla 89, Strom 88, Strom 85, Wang 92] avoid the need for frequent checkpointing by logging messages received by a process for possible replay during recovery. Each process is also occasionally checkpointed, but there is no coordination between the checkpointing of individual processes. The messages are logged *asynchronously*, saving copies of received messages in a buffer in memory and only writing the contents of the buffer to stable storage later, for example once the buffer approaches full. This buffering reduces the number of stable storage operations required and avoids the need to write partially full disk blocks to stable storage when the average message size is much less than the size of a disk block. If a process fails, it can be recovered by restarting it from some checkpoint and replaying to it the sequence of messages originally received after that checkpoint. Since some received messages may not yet have been logged, though, some surviving processes may also need to be rolled back in order to restore a consistent system state.

However, this buffering of received messages before logging may also significantly delay committing output to the outside world. In existing optimistic logging systems, the logging of a process's received messages operates independently from the need of that process or any other process to commit output. The output must wait until enough other processes have independently logged enough other messages to allow the output to be committed. Due to the dependencies between processes and the independent nature of the message logging progress of each process, any output message may be delayed for long periods before being committed. The output commit delay may be further increased by any delays in propagating knowledge of each process's logging or checkpointing progress to other processes, as needed in order to decide that the output may safely be committed. In existing systems, processes are not aware that their delays in logging messages or in propagating status information may be preventing another process from committing output.

This paper presents the design of a new transparent rollback-recovery system that adds very little overhead to distributed application programs and efficiently supports the quick commit of all output to the outside world. Each process can independently choose either to use checkpointing alone (as in consistent checkpointing methods) or to use optimistic message logging for recording its own state on stable storage, and can change this decision over the course of its own execution. For example, a process in which all execution between received messages is known to be deterministic may use either method, based on which it perceives to be the least expensive at any particular time. A process currently receiving a large number of messages may choose instead to record a new checkpoint when needed, eliminating the overhead of writing these messages to stable storage, while a process that receives few messages but has a large address space may decide to log these few messages and postpone taking an expensive checkpoint. If one or more processes in the computation are known to always be nondeterministic, they would use only checkpointing, while processes that are deterministic most of the time but occasionally experience detectable nondeterministic events may choose to use message logging during deterministic periods. As such, the system described in this paper can be viewed either as a consistent checkpointing system or as an optimistic message logging system. The design represents a balance between striving for very low overhead during failure-free execution (particularly during computational periods during which little or no output is sent to the outside world) and still being able to commit each new output message quickly (even during periods during which many output messages may be sent).

The system is based on a distributed *commit algorithm* that can be invoked at any time by a process to commit its own output, and which requires communication with and information about the minimum number of other processes in the system. The commit algorithm in effect dynamically establishes a new "recovery line" after the output was sent. The commit algorithm can also be used to inexpensively limit the number of checkpoints or logged messages that a process must retain on stable storage, or to limit the amount of rollback that a process may be forced to do in case of any future failure of any process. Multiple concurrent executions of the algorithm by different processes do not interfere with each other. The system supports the recovery of deterministic as well as nondeterministic processes, and can recover from any number of process failures, including failures during the execution of the algorithm.

In Section 2 of this paper, the assumptions made about the underlying distributed system are described. Section 3 presents some preliminary definitions used in the rest of the paper. Section 4 presents the commit algorithm and describes its use, and in Section 5, the failure detection and recovery procedure used in the system is described. Section 6 compares the system to existing message logging methods and existing checkpointing algorithms, and Section 7 presents conclusions.

2. Assumptions

The system is composed of a number of independent processes executing on a collection of fail-stop processors [Schlichting 83]. Each process has a separate address space, although in some processes, there may be more than one independent thread executing in that address space. Each process is treated as a separate *recovery unit* [Strom 85].

The processes are connected by a communication network that is not subject to network partitions. The network may lose or reorder messages, but messages are delivered with a high probability, such that reliable message delivery can be implemented by retransmitting a message a limited number of times until an acknowledgement is received from the destination process. If no acknowledgement is received, that process is assumed to have failed.

Each process has access to some stable storage server that remains accessible to other processors after any possible failure. This may be implemented, for example, by a stable storage server on the network. A failed process can be restarted on any available processor that can access the message log and checkpoint data saved on stable storage by the failed process.

Processes communicate with each other and with the outside world only through messages. Any message sent to the outside world must be delayed until it can be safely committed. Any message received from the outside world is treated the same as a message received from a process. However, if the outside world requires reliable delivery of messages to a process, the process must send an acknowledgement to the outside world upon receipt, which is treated the same as any other message sent to the outside world. No assumption of reliable delivery may be made until this acknowledgement has been delivered to the outside world.

3. Definitions

The execution of each process is divided into a sequence of *state intervals*, which are each identified uniquely for that process by a *state interval index*, incremented at the beginning of each interval. Each time a process receives a message, it begins a new state interval. Also, if any state interval of the process since its most recent checkpoint has been *nondeterministic*, the process also begins a new state interval before sending a new message. A state interval is *deterministic* if all of the process's execution in that interval can be reproduced the same as in the original execution, by restarting the process in the state it had at the beginning of the interval; otherwise, the state interval is *nondeterministic*. Nondeterministic execution may occur within a process as a result of unsynchronized memory sharing between multiple concurrent threads in the same address space or through the receipt of asynchronous software interrupts or signals. Events causing a state interval to be nondeterministic may be either detected or known at run-time (depending on the underlying system) or some processes may be declared to be treated as always nondeterministic. Deterministic execution of a process does *not* require the arrival order of messages (possibly from different senders) to be deterministic.

If all state intervals of a process are deterministic, this definition of state intervals corresponds to that used in previous message logging systems [Johnson 89, Johnson 90, Strom 85]. For processes in which some state intervals are nondeterministic, the definition has been extended to preserve the required property that all individual states of a process within a single state interval are equivalent from the point of view of all other processes in the system. The state of a process can only be seen by another process through a message sent by that process. By starting a new state interval before the process sends a message, each individual state of the process that could be seen by another process becomes part of a separate state interval.

A process state interval is called *stable* if and only if the process can be restored to some state in this interval from the information available on stable storage. This is true either if a checkpoint is available that records the process within that state interval, or if some earlier checkpoint of the process is available, all messages received by the process since that checkpoint have been logged, and all state intervals of the process since this checkpoint are deterministic. In either case, the most recent checkpoint that makes a state interval stable is called the *effective checkpoint* for that state interval. When a process recovers to some state interval σ, the process is said to *roll back* to state interval σ and all later state intervals of that process are said to be *rolled back*.

A state interval σ of some process i is called *committable* if and only if no possible future failure of any process can cause state interval σ to be rolled back. This is true if and only if there exists some *recoverable* system state in which process i is in some state interval $\alpha \geq \sigma$ [Johnson 89, Johnson 90]. When process i learns that its own state interval σ is committable, state interval σ becomes *committed*, and process i is said to *commit* state interval σ. If some state interval σ of a process i is committable (committed), then all previous state intervals $\beta < \sigma$ of process i are also committable (committed).

Since a committable state interval σ of some process i will never be rolled back in spite of any possible future failure within the system, any output sent by process i to the outside world from any state interval $\alpha \leq \sigma$ may be committed. Likewise, if the effective checkpoint of state interval σ records process i in state interval ϵ, then any checkpoint recording the state of process i in any state interval $\alpha < \epsilon$ and any logged message received by process i that started some state interval $\beta \leq \epsilon$ can safely be removed from stable storage; if process i needs to be rolled back during some future failure recovery, the checkpoint in state interval ϵ or some later checkpoint can always be used, and only those messages received by process i after that checkpoint need be replayed during the recovery.

4. The commit algorithm

4.1. Overview

The commit algorithm is invoked by a process to commit its own state interval, and communicates with only those other processes of the distributed application program with which this process must cooperate in committing this state interval. In essence, only the processes from which this process has "recently" received messages are involved in the execution of the commit algorithm. The algorithm is executed by the system on behalf of the process; its execution is not visible to the application program and does not block the execution of any process in the system.

When some process i attempts sends output from some state interval σ, the output is first saved in a buffer in memory at process i. While the application program at process i continues to execute, process i may concurrently execute the commit algorithm to commit state interval σ, and may then commit the output to the outside world. If more output messages are sent by the same process while the algorithm is executing, the process may simply execute the algorithm once more in order to commit all output sent during the first execution. If some output message need not be committed as soon as possible, the commit algorithm need not be invoked immediately and the message can remain in the buffer in memory until committed.

The commit algorithm may also be used by a process to limit the number of checkpoints or logged messages that it is required to retain on stable storage. For example, suppose a process is willing to retain only some bounded number of checkpoints, c. When the process records a new checkpoint, if σ is the state interval index of the oldest checkpoint it is willing to retain, it can use the commit algorithm to commit state interval σ and may then delete all earlier checkpoints. This also limits the number of old logged messages that must be saved on stable storage, since all messages received before state interval σ can also be deleted. To inexpensively place a loose bound on the number of checkpoints or logged messages required, the process might use this procedure only occasionally, such as after each time the process has recorded some number of new checkpoints, d. The maximum number of checkpoints is then bounded between c and $c+d$, and all logged messages received before the oldest checkpoint can be deleted. Any process may also use the commit algorithm to explicitly limit the amount of its own execution that may be lost due to any possible future rollback, by simply invoking the algorithm for the earliest state interval for which it wants to guarantee against such a loss. Such decisions on the use of the commit algorithm can be made individually by each process, and concurrent invocations of the algorithm by separate processes do not interfere.

4.2. Data structures

Each message sent by a process is tagged with the index of the sender's current state interval, and when that message is received, the receiver then depends on this state interval of the sender. The state interval index tagging a message is included when the message is logged but is not visible to the application program.

Each process i maintains the following three data structures, which are included in the process's checkpoint and are restored when the process is restarted from its checkpoint:

- A *dependency vector*, *DV*, such that *DV*$[i]$ records process i's own current state interval index, and for each other process $j \neq i$, *DV*$[j]$ records the maximum index of any interval of process j on which process i then depends. If process i has no dependency on any state interval of some process j, then *DV*$[j]$ is set to $\perp$, which is less than all possible state interval indices. A receiving process *transitively* depends on all state intervals on which the sending process depended when the message was sent, but the dependency vector records only those state intervals on which the process *directly* depends [Johnson 89, Johnson 90]. By passing only the current state interval index of the sender rather than the full (transitive) dependency vector [Strom 85], the failure-free overhead on the message passing system is reduced.

- A *commit vector*, *COMMIT*, such that *COMMIT*$[i]$ records the index of process i's own current maximum committed state interval, and for each other process $j \neq i$, *COMMIT*$[j]$ records the maximum index of any interval of process j that process i knows is committable. If process i knows of no committable state intervals of some process j, then *COMMIT*$[j]$ is set to $\perp$.

- A *dependency vector history* that records the incremental changes to the process's dependency vector and allows previous dependency vectors of the process to be recalled. At the start of each state interval, $DV[i]$ is incremented, and only one other entry in the process's dependency vector can change. The dependency vector history records the process identifier and previous value of this entry for each state interval σ of the process such that $\sigma > COMMIT[i]$, the maximum committed state interval of the process.

A process using message logging also maintains a *volatile message log buffer* in memory in which copies of received messages are saved, but the volatile message log buffer of a process is *not* included in the process's checkpoint. Messages from the buffer may be written to stable storage at any time, logging those messages, but all logging is performed asynchronously, without blocking the execution of the process.

Each process has an assigned *computation identifier*, and all processes participating in the same distributed application program share the same computation identifier. When a process is created, the computation identifier is usually copied from the creating process, but a new computation may also be started by assigning a new unique identifier. Any process not running as part of a fault-tolerant computation is assigned a reserved computation identifier of 0. When a fault-tolerant process sends a message, if the sender and receiver have different computation identifiers, the message is treated as output to the outside world by the sender. This allows a large system to be divided into separate computations to prevent a failure in one computation from causing any process in another computation to roll back [Lowry 91, Sistla 89], and also allows fault-tolerant processes to communicate safely with non-fault-tolerant processes such as standard system servers. If both sender and receiver are on the same machine, the receiver's computation identifier is readily available to the sending operating system kernel; otherwise, the underlying system may use an address resolution mechanism such as ARP [Plummer 82] to find and cache the receiver's network address, and this mechanism may be modified to return and cache the receiver's computation identifier at the same time.

4.3. The algorithm

Theorem 1 A state interval σ of some process i is committable if and only if

- some later state interval $\alpha > \sigma$ of process i is committable, or
- state interval σ of process i is stable and for all $j \neq i$, state interval $DV[j]$ of process j is committable, where DV is the dependency vector of state interval σ of process i.

Proof (Sketch) If state interval α will never be rolled back, then an earlier state interval σ cannot be rolled back either, and thus state interval σ must be committable.

If no later state interval α is committable, then some failure might force process i to roll back as far as state interval σ. In order to be able to recover process i to state interval σ, state interval σ must therefore be stable. In order to maintain a consistent system state upon recovery of process i to state interval σ, for each other process $j \neq i$, process j cannot roll back to any state interval earlier than $DV[j]$. By the definition of a committable state interval, this implies that for each $j \neq i$, state interval $DV[j]$ of process j is committable. □

Using this theorem, state interval σ of process i could be committed by recursively attempting to commit state interval $DV[j]$ of each other process j. However, this simplistic algorithm would result in a large number of redundant messages and invocations of the algorithm. The top-level execution of the algorithm may send up to one request to each process in the system to commit some state interval of that process. For each of these requests, the process at which it executes may send an additional request to each process in the system, which each in turn may also send request to each process, and so on, recursively sending additional requests until each branch of the recursion independently encounters a state interval that has already been committed. For each process j, however, only the maximum state interval of any of the intervals for which this simplistic implementation would invoke the algorithm is important; if this state interval is committable, then by definition, all earlier state intervals of the same process are also committable.

The commit algorithm shown in Figure 1 avoids many of these redundant requests, by traversing the (transitive) dependencies of state interval σ (as implied by Theorem 1) in a series of *rounds*, such that in each round, at most *one* request is sent to each process. A request is sent to each process only for the *maximum* state interval index then needed in the traversal. Each round may thus send only $O(n)$ requests, for a system of n processes. The traversal is acyclic, and thus at most $O(n)$ rounds may be required, although the actual number of rounds performed will typically be less than this, depending on the communication pattern between processes of the application program. The number of messages required is further reduced by using the *COMMIT* vector of a process to record the state intervals of each process that are known by that process to be committable, thus avoiding many requests before sending them.

The main procedure, *COMMIT_STATE*, is called with the single parameter σ, the index of the state interval of the local process to be committed, and the procedures *NEED_STABLE* and *NEW_COMMIT* are executed as remote procedure calls at other processes by *COMMIT_STATE*. The vector *TDV* collects the transitive dependencies of state interval σ, and the vector *NEED* records those entries of *TDV* from which the dependency traversal is still needed. During the execution of *COMMIT_STATE* by some process i, each *NEED_STABLE* request to a process j instructs process j to find an existing stable state interval $\alpha \geq \delta$ of process j or to make one stable by message logging or checkpointing. If process j has already committed state interval δ, it replies "committed" and returns its own current *COMMIT* vector, which process i then merges into its local *COMMIT* vector. If state interval δ has not already been committed, process j has two choices: it may use any state interval $\alpha \geq \delta$ that is already stable (and return "stable" from *NEED_STABLE*) or may decide to asynchronously make some state interval $\alpha \geq \delta$ stable by checkpointing or message logging (and return "volatile" from *NEED_STABLE*). The return from *NEED_STABLE* also includes the dependency vector for the chosen state interval α, which process j can recall using its dependency vector history. If process j returns "volatile," the necessary logging or checkpointing is done in parallel with the further execution of the process and with the execution of the commit algorithm, and process j later sends a "done" message to process i when this logging or checkpointing completes. The vector *VOLATILE* in *COMMIT_STATE* records whether the corresponding state interval in *TDV* was volatile when the last call to *NEED_STABLE* for that process returned.

The algorithm allows a great deal of flexibility in the particular state interval α chosen to satisfy a *NEED_STABLE* request. For processes using optimistic message logging, the process would normally choose to log any messages from its volatile message log buffer that are necessary to make state interval δ stable, but the process may instead choose to use any later checkpointed state in order to avoid logging the earlier messages. Processes may also choose to use only checkpointing (as in consistent checkpointing systems), for example, if they are currently receiving a very large number of messages (that would otherwise need to be logged) or have executed nondeterministically since their previous checkpoint. Such processes would instead record a new process checkpoint in their current state interval and use this state interval as α. On the other hand, for most processes, it may be quicker to log messages than to record a new checkpoint since the amount of data to be written to stable storage will be less (the stable storage writes must *finish*

```
procedure COMMIT_STATE(σ)
  if σ ≤ COMMIT[pid] then return;
  TDV ← dependency vector for state interval σ;
  for all j do
    NEED[j] ← ⊥;
    if TDV[j] > COMMIT[j] then NEED[j] ← TDV[j];
  for all j do VOLATILE[j] ← false, ACTIVE[j] ← false;
  INITIATOR[pid] ← true;
  while ∃ j such that NEED[j] ≠ ⊥ do        /* loop again while another round is needed */
    for all j such that NEED[j] ≠ ⊥ do in parallel        /* send all requests for this round in parallel */
      (status, UPDATE) ← NEED_STABLE(NEED[j]) at process j;
      if status = “committed” then
        for all k do
          if UPDATE[k] > COMMIT[k] then
            COMMIT[k] ← UPDATE[k];
            if COMMIT[k] ≥ TDV[k] then NEED[k] ← ⊥;
        if σ ≤ COMMIT[pid] then
          INITIATOR[pid] ← false;
          return;
      else        /* status is “stable” or “volatile” */
        ACTIVE[j] ← true;
        VOLATILE[j] ← (status = “volatile”);
        for all k do
          if UPDATE[k] > TDV[k] then
            TDV[k] ← UPDATE[k];
            if TDV[k] > COMMIT[k] then NEED[k] ← TDV[k];
        if NEED[j] = UPDATE[j] then NEED[j] ← ⊥;
  for all j such that VOLATILE[j] do wait for “done” from process j;
  INITIATOR[pid] ← false;
  for all j do COMMIT[j] ← max(COMMIT[j], TDV[j]);
  for all j such that ACTIVE[j] do NEW_COMMIT(COMMIT) at process j;
  return;

function NEED_STABLE(δ)
  if δ ≤ COMMIT[pid] then return (“committed”, COMMIT);
  INITIATOR[sender] ← true;
  if state interval δ of this process is stable then
    return (“stable”, dependency vector of state interval δ);
  α ← minimum state interval index such that α ≥ δ and α is stable or can be made stable;
  if state interval α is stable then
    return (“stable”, dependency vector of state interval α);
  else
    asynchronously begin making state interval α stable and reply “done” when complete;
    return (“volatile”, dependency vector of state interval α);

procedure NEW_COMMIT(UPDATE)
  INITIATOR[sender] ← false;
  for all j do COMMIT[j] ← max(COMMIT[j], UPDATE[j]);
  return;
```

Figure 1 The commit algorithm at process *pid* before the state interval can be committed), and thus the system allows those processes to avoid the more expensive checkpointing. By choosing the minimum such $\alpha \geq \delta$ (subject to the decisions as to whether to use message logging or checkpointing), all components of the dependency vector for this state interval α are minimized, reducing the further dependencies that must be traversed by process i in *COMMIT_STATE*.

Each process maintains a boolean vector *INITIATOR*, such that for all i, *INITIATOR*$[i]$ is true if and only if this process is currently participating in the execution of the commit algorithm invoked by process i, and *COMMIT_STATE* at process i maintains a boolean vector *ACTIVE*, such that for all j, *ACTIVE*$[j]$ is true if and only if this execution of *COMMIT_STATE* has sent a *NEED_STABLE* request to process j that resulted in process j setting its *INITIATOR*$[i]$ to true. Define the boolean value *committing* for a process to be true if and only if there exists some entry in the *INITIATOR* vector for that process that is currently true. All application program messages sent by a process are tagged with the current *committing* value for that process. When a process receives a message, if the value tagging the message and the receiver's *committing* value are both true, then the message is delayed and not delivered to the application program at the receiver process until the completion of all executions of the algorithm in which this receiver is currently participating; the message is then delivered when the receiver changes its last *INITIATOR* entry back to false and thus changes its *committing* value to false. This inhibition in receiving certain messages during the execution of the algorithm is necessary in order to avoid any possibility of livelock in the execution of the algorithm. If a *NEED_STABLE* request at some process returns some state interval $\alpha > \sigma$ (for example, by recording a new checkpoint in the process's current state interval α), state interval α may have dependencies beyond those of state interval δ (some components of the dependency vector for state interval α may be greater than the corresponding components of the dependency vector for state interval δ). In the unlikely case in which this continues with future calls to *NEED_STABLE* as the system continues to execute in parallel with the algorithm, the algorithm could continue execution indefinitely without terminating. By preventing a "committing" process from receiving a message sent from another "committing" process, no cyclic propagation of such dependencies is possible and thus no livelock is possible. At the completion of *COMMIT_STATE*, process i sends a *NEW_COMMIT* request to each process j for which *ACTIVE*$[j]$ is true, distributing its new *COMMIT* vector and allowing process j to change its *INITIATOR*$[i]$ back to false.

The algorithm also incorporates a number of additional optimizations:

- During a round of the algorithm, all *NEED_STABLE* requests indicated by the current value of *NEED* are sent in parallel, allowing concurrent execution of all *NEED_STABLE* calls within each round.
- If provided by the network hardware, a single unreliable broadcast can be used in each round to broadcast all parallel *NEED_STABLE* requests for that round in a single message. After collecting replies from the *NEED_STABLE* broadcast during some small timeout period, any missing replies can be ignored, since any missing *NEED_STABLE* request for some particular process j will effectively be retransmitted on the next round of the algorithm if needed.
- The use of parallel *NEED_STABLE* requests also allows any stable storage operations required by the algorithm to overlap as much as possible.

The correctness of the algorithm is established by Theorem 1 and by the above discussion. This correctness can also be shown using the model of dependency matrices and the system history lattice [Johnson 89, Johnson 90], but this proof is omitted here due to space considerations. No deadlock is possible in the execution of the commit algorithm, since the algorithm executes independently of the application program at each process. Multiple concurrent executions of the algorithm by different processes do not interfere with each other, since the set of recoverable system states forms a lattice [Johnson 89, Johnson 90]. Changes to a process's commit vector made by one execution of the algorithm are valid in all other concurrent executions and are immediately visible to those executions.

To illustrate the operation of the commit algorithm, Figure 2 shows the messages exchanged during one execution. Process 2 invokes the algorithm to commit its current state interval, σ, and sends two rounds of *NEED_STABLE* requests. In this example, two processes return a "volatile" status: process 5 begins asynchronously logging some messages from its volatile message log buffer to stable storage, and process 3 chooses instead to begin recording a new checkpoint of itself. The time during which processes 3 and 5 are performing stable storage operations is indicated in Figure 2 by shading along the lines representing the execution of each process; both processes continue to execute in parallel with these stable storage operations. Once the "done" message is received from each process by process 2 and the algorithm completes, state interval σ of process 2 has been committed. All five processes continue to execute in parallel with the execution of the algorithm.

5. Failure detection and recovery

Process failures are detected through a timeout mechanism. As mentioned in Section 2, if no acknowledgement of a message being reliably transmitted by the underlying system is received after a bounded number of retransmissions, that process is assumed to have failed and must be rolled back to a stable state interval. When a process rolls back, any other process that depends on some rolled back state interval of that process

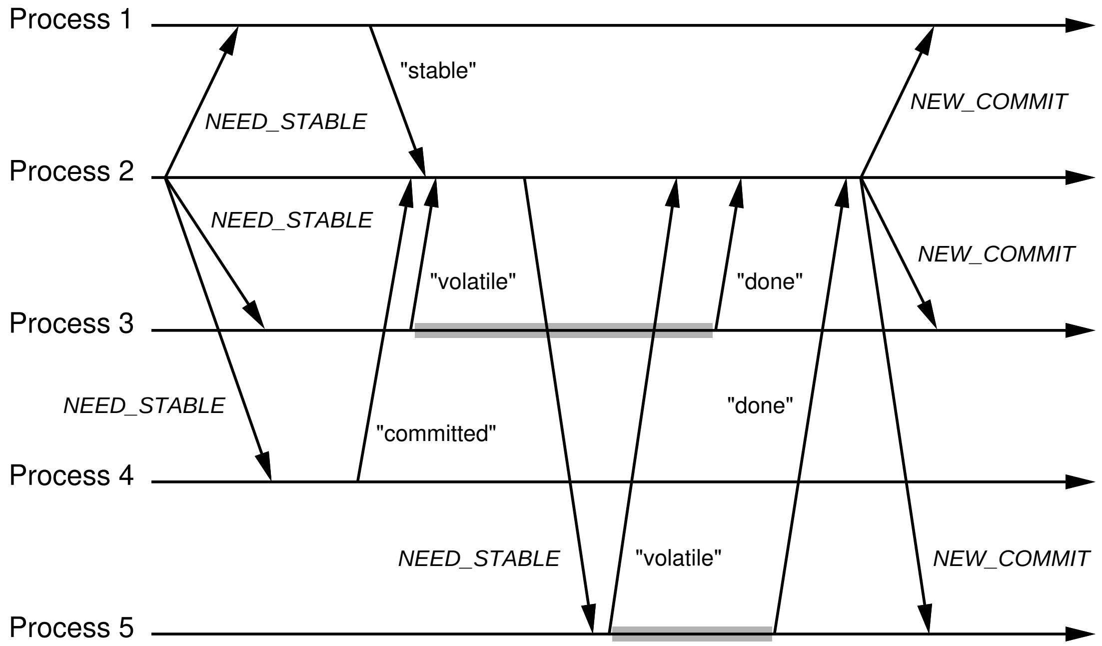

Figure 2 Example execution of the commit algorithm by process 2

is called an *orphan process* [Strom 85] due to that roll back. An orphan process must also be rolled back to some earlier state that does not yet depend on any rolled back state interval of any process.

When any process rolls back, notification of the rollback is sent to all processes in the distributed application program. The stable storage server from which the checkpoint for this rollback will be read is first notified as part of the recovery. That server reliably notifies any other stable storage servers in the system, and the stable storage servers cooperate to provide a total ordering on all rollbacks in the program using any ordered group communication protocol among the stable storage servers [Birman 87, Chang 84, MelliarSmith 90, Rodrigues 92]. Each stable storage server then sends notifications of the rollback to each process in the distributed application program that currently has checkpoints stored with that server, as determined by the process's computation identifier recorded in each checkpoint (Section 4.2). The notifications for concurrent or closely spaced rollbacks of different processes may be combined into a single notification message. The notifications to each process should be sent with high probability of delivery but need not be sent reliably. If an unreliable broadcast capability is provided by the network hardware, a single broadcast may be used to send the rollback notification to all processes.

For each rollback of any process, the distributed application program enters a new *incarnation*, identified by a new *computation incarnation number*, which is simply a total count of process rollbacks that have occurred within the application program. The new incarnation number is included in each rollback notification message, and thus also forms a sequence number on these messages. When a process failure is detected, it is possible that the process has actually not failed but for some reason has been temporarily unable to send or receive messages. Such a process will be recovered (a new version of it started), even though the original version of the process is still executing. The computation incarnation number is used to make any surviving old versions of such a process harmless and to eventually terminate those old versions.

A failed process rolls back to its most recent state interval that is stable, based on its checkpoints and logged messages that had been recorded on stable storage before its failure. The notification for the rollback of some process k includes the identification of the process that is rolling back, the index σ of the state interval to which process k is rolling back, and the new incarnation number of the computation. When a process receives a rollback notification, it remembers the new computation incarnation number, and then checks its dependency vector to determine if it has become an orphan due to this rollback. Process i has become an orphan due to this rollback of process k to state interval σ if $DV[k] > \sigma$, where DV is the current dependency vector of process i. In this case, process i then rolls back to its most recent state interval α for which element k in the dependency vector for state interval α is less than σ. The notification of this rollback of process i is sent to all processes in the same way as the notification of the original rollback from the failure. No domino effect is possible, since no process can be forced to roll back beyond its most recent committed state interval.

Every message sent within the application program is tagged with the current computation incarnation number as known by the sending process. When received by another process within the application program, if the incarnation number tagging the message matches the current computation incarnation number as known by the receiver, the message is accepted. If, instead, the message's incarnation number is less than the receiver's, the receiver rejects the message and returns an indication of this rejection to the sender. The sender must have missed one or more rollback notifications. The sender checks with any stable storage server for the correct current incarnation number of its computation and for copies of any missing rollback notifications, and handles these notifications in the same way as any other rollback notification. However, if one of these notifications announces the sender's own rollback, the sender must have been assumed to have failed earlier and has been recovered elsewhere. In this case, this copy of the sender process (the "obsolete" copy) simply terminates execution. Otherwise, the sender retransmits the original message with the correct computation incarnation number. Similarly, if the receiver's incarnation number is less than the incarnation number tagging the message received, the receiver checks with any stable storage server to determine the correct current incarnation number of its computation and to retrieve a copy of any missing rollback notifications, and then handles these notifications as described above.

If a process is executing the commit algorithm when it receives a rollback notification, the algorithm is terminated and restarted, but any modifications made to the process's *COMMIT* vector during the algorithm remain valid and are retained. The algorithm restarts at the beginning of *COMMIT_STATE*. If a process receives a rollback notification for some other process i and *INITIATOR*$[i]$ for that process currently true, then the process changes *INITIATOR*$[i]$ back to false. A failed process can respond to *NEED_STABLE* requests as soon as its recovery has started to the point that it has access to its checkpoints and message logs on stable storage.

No livelock is possible in the execution of the distributed application program as a result of failures and recoveries occurring in the system. As described by Koo and Toueg [Koo 87], without synchronization among processes during recovery, a single failure could cause an infinite number of process rollbacks. By an inconsistent interleaving of application program messages and process rollback notifications at different processes for a series of rollbacks, it is possible to indefinitely continue a pattern of rollbacks among a group of processes such that each process rolls back and then begins to execute forward again, only to be forced to roll back again due to the rollback of some other process. The processes continue to execute, but the system makes no progress. The use of the computation incarnation number prevents such a livelock. By placing a total order on all process rollbacks within the distributed application program and by including the current incarnation number in each message sent, each process is forced to handle each rollback notification in a consistent order relative to the messages being sent by the application program.

6. Comparison to related work

6.1. Message logging methods

Pessimistic message logging protocols log messages *synchronously* [Borg 83, Borg 89, Johnson 87, Powell 83]. The protocol guarantees that any failed processes can be recovered individually without affecting the states of any processes that did not fail, and prevents processes from proceeding until the logging that is required by this guarantee has been completed. This is achieved either by blocking a process receiving a message until the message is logged [Borg 83, Borg 89, Powell 83], or by blocking the receiver if it attempts to send a new message before all messages it has received are logged [Johnson 87]. No special protocol is required for output commit, and thus, output commit need not be delayed. However, the overhead of logging all messages synchronously slows the system down even when no output is being sent.

Optimistic message logging protocols, instead, log messages *asynchronously* [Johnson 89, Johnson 90, Juang 91, Lowry 91, Sistla 89, Strom 88, Strom 85, Wang 92], and were developed to avoid the overhead of synchronous logging [Strom 85]. However, any output message may be significantly delayed, since the message logging of each process operates independently and may be delayed arbitrarily long, and also because processes may delay the propagation of information on their logging or checkpointing progress that is necessary to determine that the output may be committed. In addition, the algorithms used for output commit in previous systems compute much more information about the entire system than is needed by any process for committing its own output. For example, the exact state of each other process in the current maximum recoverable state of the system [Johnson 89, Johnson 90, Sistla 89] or the exact number of message logged by each other process in the system [Strom 85] is of no concern to any individual process and is not necessary for output commit. The commit algorithm presented in this paper computes only the information necessary for committing a state interval, and is unique in directly supporting the quick commit of all output to the outside world. Unlike previous optimistic message logging systems, the system presented here also supports the recovery of nondeterministic processes through checkpointing, while still allowing deterministic processes to use message logging to reduce their own failure-free overhead.

The rollback-recovery algorithm of the Manetho system [Elnozahy 92b] also attempts to commit all output quickly to the outside world, while keeping failure-free overhead low. Manetho uses an *antecedence graph*, together with a form of *sender-based message logging* in volatile memory [Johnson 87], to allow processes to commit output without coordination with other processes. The graph records the dependencies between processes, and updates to the graph are appended to the messages that a process sends, in order to propagate portions of the graph to other processes. Manetho can commit output more quickly than the commit algorithm presented here, but its failure-free overhead is higher due to the need to maintain the antecedence graph and to propagate graph updates. As such, Manetho represents a different point in the tradeoff between output commit latency and failure-free overhead. Manetho also offers only restricted support for nondeterministic processes in the system.

6.2. Checkpointing algorithms

Many *consistent checkpointing* algorithms require all processes in the system to record a new checkpoint each time a new consistent checkpoint is made [Briatico 84, Chandy 85, Cristian 91, Elnozahy 92a, Kaashoek 92, Lai 87, Li 91, Moura e Silva 92, Spezialetti 86, Ramanathan 88, Tong 89]. Since a new consistent checkpoint must be recorded each time before committing output to the outside world, these systems may experience very high overhead in periods during which many new output messages are sent. Although each process checkpoint can be written in parallel with the execution of the application program [Elnozahy 92a, Johnson 89, Li 90], each output commit must nonetheless wait until all data for the new consistent checkpoint has been written to stable storage. For systems of many processes or those containing processes with very large address spaces, this may further delay output commit.

Koo and Toueg proposed a consistent checkpointing algorithm [Koo 87] in which only the minimum number of processes are required to record a new checkpoint in order to establish a new (global) consistent checkpoint. Their algorithm, though, requires FIFO channels between processes, whereas the commit algorithm presented here makes no such assumption. Their algorithm also prevents the application program at a process from sending any messages between the beginning of the algorithm at that process and the completion of the entire algorithm across all processes. In contrast, the commit algorithm here interferes much less with the communication of the application program, as processes are prevented from receiving (rather than sending) a message during the execution of the algorithm *only* for particular messages whose receipt could prolong the execution of the algorithm and possibly lead to a livelock in its execution. Without this restriction, the algorithm would still record a consistent system state, but could take one or more additional rounds to converge.

The protocol of Venkatesh et al [Venkatesh 87] also requires FIFO channels. In addition, it records more separate process checkpoints than the commit algorithm presented here, since it uses each process checkpoint in only one recovery line and thus must record a separate process checkpoint for each consistent system state that may include a given state of a process. The protocol of Israel and Morris [Israel 89] does not inhibit sending or receiving messages during the execution of the algorithm, but does require FIFO channels. Leu and Bhargava have proposed a protocol [Leu 88] that does not require FIFO channels. The basic version of their algorithm inhibits processes sending messages during the algorithm, as in Koo and Toueg's algorithm, but an extension to the algorithm avoids any inhibition. However, their algorithm is more complicated than the commit algorithm presented here and may require many more messages than the commit algorithm, since each process in their checkpoint tree must send a request to each *potential child*, many of which may be rejected when the child does not become a *true child* because it has already been included in the checkpoint tree from a different parent. By collecting the replies at the initiator on each round, the commit algorithm here avoids many of these redundant messages.

Bhargava and Lian have proposed a protocol [Bhargava 88] that uses only checkpointing for recording the states of processes, but in which processes record their checkpoints independently without the coordination present in consistent checkpointing systems. The goal of their system is to avoid the overhead of the coordination protocol. When a failure later occurs, they instead attempt to find some set of process checkpoints on stable storage that happens to record a consistent state of the system, which can be used in recovery. Although this may allow failure-free overhead to be very low, their system cannot support the quick commit of output to the outside world for the same reasons as discussed above for optimistic message logging systems. In addition, there is no guarantee that any output can be committed even after recording an arbitrarily large number of new process checkpoints, since there is no guarantee that any of these checkpoints form part of a consistent system state with the other available checkpoints of other processes.

7. Conclusion

Existing rollback-recovery methods using consistent checkpointing may cause *high overhead* for programs that frequently send output to the outside world, whereas existing methods using optimistic message logging may cause *large delays* in committing output. The design presented in this paper addresses these problems, adding very little overhead to the failure-free execution of the system while still being able to quickly commit all messages sent to the outside world, and thus represents a balance in the tradeoff between failure-free overhead and output commit latency inherent in any transparent rollback-recovery system.

The system is based on a distributed *commit algorithm* that can be used by a process to commit its own output, as well as to inexpensively limit the number of checkpoints or logged messages that the process must retain on stable storage, or the amount of rollback that the process may be forced to do in case of any possible future failure of any process. Each process can independently choose to use either checkpointing or optimistic message logging for recording its own state as needed on stable storage, and can change this decision over the course of its own execution. This choice allows each process to use the method most suited to its own particular circumstances. The system supports recovery of deterministic as well as nondeterministic processes, and can recover from any number of process failures, including failures during the execution of the algorithm.

Acknowledgements

I would like to thank Mootaz Elnozahy, Wayne Sawdon, Po-Jen Yang, Matt Zekauskas, Willy Zwaenepoel, and the anonymous referees for their valuable comments on the content and presentation of this paper.

References

[Bhargava 88] Bharat Bhargava and Shy-Renn Lian. Independent checkpointing and concurrent rollback for recovery—An optimistic approach. In *Proceedings of the Seventh Symposium on Reliable Distributed Systems*, pages 3–12. IEEE Computer Society, October 1988.

[Birman 87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[Borg 83] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 90–99. ACM, October 1983.

[Borg 89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[Briatico 84] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, pages 207–215, October 1984.

[Chandy 85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[Chang 84] Jo-Mei Chang and N. F. Maxemchuck. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Cristian 91] Flaviu Cristian and Farnam Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 12–20. IEEE Computer Society, September 1991.

[Elnozahy 92a] Elmootazbellah Nabil Elnozahy, David B. Johnson, and Willy Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47. IEEE Computer Society, October 1992.

[Elnozahy 92b] Elmootazbellah Nabil Elnozahy and Willy Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, C-41(5):526–531, May 1992.

[Israel 89] Stephen Israel and Derek Morris. A non-intrusive checkpointing protocol. In *Eighth Annual International Phoenix Conference on Computers and Communications: 1989 Conference Proceedings*, pages 413–421. IEEE Computer Society, March 1989.

[Johnson 87] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *The Seventeenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 14–19. IEEE Computer Society, June 1987.

[Johnson 89] David B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. Ph.D. thesis, Rice University, Houston, Texas, December 1989.

[Johnson 90] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, September 1990.

[Juang 91] Tony T-Y. Juang and S. Venkatesan. Crash recovery with little overhead (preliminary version). In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 454–461. IEEE Computer Society, May 1991.

[Kaashoek 92] M. F. Kaashoek, R. Michiels, H. E. Bal, and A. S. Tanenbaum. Transparent fault-tolerance in parallel Orca programs. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems III*, pages 297–312. USENIX, March 1992.

[Koo 87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[Lai 87] Ten H. Lai and Tao H. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, May 1987.

[Leu 88] Pei-Jyun Leu and Bharat Bhargava. Concurrent robust checkpointing and recovery in distributed systems. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 154–163. IEEE Computer Society, February 1988.

[Li 90] Kai Li, Jeffrey F. Naughton, and James S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming*, pages 79–88. ACM, March 1990.

[Li 91] Kai Li, Jeffrey F. Naughton, and James S. Plank. Checkpointing multicomputer applications. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 2–11. IEEE Computer Society, September 1991.

[Lowry 91] Andy Lowry, James R. Russell, and Arthur P. Goldberg. Optimistic failure recovery for very large networks. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 66–75. IEEE Computer Society, September 1991.

[MelliarSmith 90] P. M. Melliar-Smith, Louise E. Moser, and Vivek Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[Moura e Silva 92] Luis Moura e Silva and João Gabriel Silva. Global checkpointing for distributed programs. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 155–162. IEEE Computer Society, October 1992.

[Plummer 82] David C. Plummer. An Ethernet address resolution protocol. Internet Request For Comments RFC 826, November 1982.

[Powell 83] Michael L. Powell and David L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 100–109. ACM, October 1983.

[Ramanathan 88] P. Ramanathan and K. G. Shin. Checkpointing and rollback recovery in a distributed system using common time base. In *Proceedings of the Seventh Symposium on Reliable Distributed Systems*, pages 13–21. IEEE Computer Society, October 1988.

[Rodrigues 92] Luís Rodrigues and Paulo Verissimo. xAMp: a multi-primitive group communications service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 112–121. IEEE Computer Society, October 1992.

[Schlichting 83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant distributed computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[Sistla 89] A. Prasad Sistla and Jennifer L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 223–238. ACM, August 1989.

[Spezialetti 86] Madalene Spezialetti and Phil Kearns. Efficient distributed snapshots. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 382–388. IEEE Computer Society, May 1986.

[Strom 85] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[Strom 88] Robert E. Strom, David F. Bacon, and Shaula A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *The Eighteenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 44–49. IEEE Computer Society, June 1988.

[Tong 89] Zhijun Tong, Richard Y. Kain, and W. T. Tsai. A low overhead checkpointing and rollback recovery scheme for distributed systems. In *Proceedings of the Eighth Symposium on Reliable Distributed Systems*, pages 12–20. IEEE Computer Society, October 1989.

[Venkatesh 87] K. Venkatesh, T. Radhakrishnan, and H. F. Li. Optimal checkpointing and local recording for domino-free rollback recovery. *Information Processing Letters*, 25(5):295–303, July 1987.

[Wang 92] Yi-Min Wang and W. Kent Fuchs. Optimistic message logging for independent checkpointing in message-passing systems. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 147–154. IEEE Computer Society, October 1992.