Output-Driven Distributed Optimistic Message Logging and Checkpointing

David B. Johnson
Willy Zwaenepoel

Rice COMP TR90-118

May 1990

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251-1892

(713) 527-4834

This research was supported in part by the National Science Foundation under grants CCR-8716914 and CDA-8619893, and by the Office of Naval Research under contract ONR N00014-88-K-0140.

Abstract

Although *optimistic* fault-tolerance methods using *message logging and checkpointing* have the potential to provide highly efficient, transparent fault tolerance in distributed systems, existing methods are limited by several factors. Coordinating the asynchronous message logging progress among all processes of the system may cause significant overhead, limiting their ability to scale to large systems and offsetting some of the performance gains over simpler *pessimistic* methods. Furthermore, logging all messages received by each process may place a substantial load on the network and file server in systems with high communication rates. Finally, existing methods do not support nondeterministic process execution, such as occurs in multithreaded processes and those that handle asynchronous interrupts.

This paper presents a new method using optimistic message logging and checkpointing that addresses these limitations. Any fault-tolerance method must delay output from the system to the *outside world* until it can guarantee that no future failure can force the system to roll back to a state before the output was sent. With this new method, only this need to *commit output* forces any process to log received messages or to checkpoint. Each process commits its own output, with the cooperation of the minimum number of other processes, and any messages not needed to allow pending output to be committed need not be logged. Individual processes may also dynamically switch to checkpointing *without* message logging, to avoid the expense of logging a large number of messages or to support their own nondeterministic execution.

1 Introduction

Optimistic fault-tolerance methods using *message logging and checkpointing* [Strom85, Johnson88, Sistla89] have the potential to provide highly efficient, transparent fault tolerance in distributed systems. Messages received by a process are logged *asynchronously* on stable storage, without delaying the execution of the process. By removing the synchronization between computation and message logging required by *pessimistic* methods [Powell83, Borg83, Johnson87, Borg89], optimistic message logging can improve performance in the absence of failures. Although failure recovery with optimistic methods is more complex than with pessimistic methods, failures in the system are generally infrequent.

However, existing methods using optimistic message logging and checkpointing are limited by several factors. Any fault-tolerance method must delay output from the system to the *outside world* until it can guarantee that no future failure can force the system to roll back to a state before the output was sent. The outside world consists of all objects with which processes interact that do not participate in the recovery protocols of the system, such as the user's display terminal or a hardware time-of-day clock. Determining when output may be *committed* to the outside world with optimistic message logging requires some coordination of the message logging progress of all processes in the system, which may cause significant overhead and offset some of the performance gains over simpler pessimistic methods. This coordination also limits the ability of these methods to scale to large systems. Furthermore, logging all messages received by each process on stable storage may place a substantial load on the network and file server in systems with high communication rates. Finally, existing methods do not support nondeterministic process execution, such as occurs in multithreaded processes sharing memory and those that handle asynchronous interrupts.

Output-driven optimistic message logging and checkpointing is a new method that addresses these limitations. Only the need to commit output forces any process to log received messages or to checkpoint. Each process commits its own output, with the cooperation of the minimum number of other processes, as determined by the dependencies between processes. Any messages not needed to allow pending output to be committed need not be logged. Individual processes may also dynamically switch between using either message logging and checkpointing or using checkpointing alone. This allows processes to avoid the expense of logging a large number of received messages, and allows processes to support their own nondeterministic execution. During periods in which a process executes nondeterministically, it uses only checkpointing, but during periods of deterministic execution, it may also use message logging to reduce the number of checkpoints required. These decisions are entirely local to each process.

Previous methods using optimistic message logging and checkpointing [Strom85, Johnson88, Sistla89] attempt to maintain current knowledge of the state to which the system would be recovered if a failure were to occur. Any output sent to the outside world before this state may be committed. However, maintaining this knowledge requires coordination between between all processes in the system, and can significantly increase the fault-tolerance overhead. Furthermore, existing methods must continuously update this information, even during periods in which no output is sent from the system. Output-driven message logging and checkpointing avoids this unnecessary overhead. The

algorithm allows all output to be committed quickly after being sent, while reducing the overhead required to determine when output can be committed.

This paper presents the motivation and design of output-driven optimistic message logging and checkpointing. Section 2 describes the distributed system assumptions made in this design. The theoretical development of this design is based on our previous model for reasoning about message logging and checkpointing methods [Johnson88, Johnson89], which is reviewed in Section 3. In Section 4, the model is extended to define a *committable state interval*, and Section 5 presents the design of the output-driven message logging and checkpointing method based on this definition. Related work is considered in Section 6, and conclusions are presented in Section 7.

2 Distributed System Assumptions

The design of output-driven message logging and checkpointing depends on the following assumptions of the underlying distributed system:

- The system consists of an asynchronous network of fail-stop processors [Schlichting83].
- Processes communicate with one another only through messages.
- Any process can send a message to any other process, regardless of their locations within the network.
- Packet delivery on the network need not be guaranteed, but reliable delivery of a packet can be achieved by retransmitting it a bounded number of times until an acknowledgement arrives from its destination.
- Each process has available some form of stable storage that is always accessible after any failure. Processes need not share a common stable storage service.
- The execution of each process is deterministic between received messages. The state of a process is thus completely determined by its starting state and by the sequence of messages it has received. This assumption is relaxed in Section 5.6 to allow arbitrary *nondeterministic* execution of any processes.

3 Review of the Model

This work is based on the system model developed as part of our earlier work with optimistic message logging and checkpointing [Johnson88, Johnson89]. The model is independent of the message logging protocol used by the underlying system, and is based on the notion of *dependency* between the states of processes that results from communication in the system. This section provides a brief review of that model.

In the model, the execution of each process is divided into a sequence of *state intervals* by the messages that the process receives, and execution of the process within each state interval is assumed to be deterministic. That is, each time a process receives a message, it begins a new state

interval, in which the execution of the process is a deterministic function of the contents of the message and the state of the process at the time that it received the message. Each state interval of a process is identified by a unique *state interval index*, which is a count of messages received by the process.

When one process receives a message from another process, the new state interval of the receiver depends on the state interval of the sender from which the message was sent, since any part of the sender's state may have been included in the message. The current set of dependencies of a process are represented by a *dependency vector*

$$\langle \delta_* \rangle = \langle \delta_1, \delta_2, \delta_3, \ldots, \delta_n \rangle \ ,$$

where n is the total number of processes in the system. In the dependency vector for each process i, each component j, $j \neq i$, records the maximum index of any state interval of process j on which process i currently depends. Component i in process i's dependency vector records the index of process i's current state interval. If a process i has no dependency on any state interval of some process j, component j is set to $\perp$, which is less than all possible state interval indices. The dependency vector records only the *direct* dependencies of the process, resulting from the receipt of a message sent from that state interval in the sending process.

A process state interval is called *stable* if and only if all messages received by the process since its previous checkpoint have been logged. This previous checkpoint for each stable process state interval is called the *effective checkpoint* for that state interval. Only a stable process state interval can be recreated from information on stable storage during recovery. When a process is created, it is checkpointed before beginning execution, and thus state interval 0 is stable for each process. Processes may log messages or record new checkpoints at any time, without coordination between processes, and may record new checkpoints without logging some previously received messages. Thus, the set of stable state intervals of a process need not be consecutive, but each consecutive group of stable state intervals begins with a state interval recorded in a checkpoint.

A collection of process states, one for each process in the system, defines a system state, which is represented by an $n \times n$ *dependency matrix*

$$\mathbf{D} = [\delta_{**}] = \begin{bmatrix} \delta_{1\,1} & \delta_{1\,2} & \delta_{1\,3} & \ldots & \delta_{1\,n} \\ \delta_{2\,1} & \delta_{2\,2} & \delta_{2\,3} & \ldots & \delta_{2\,n} \\ \delta_{3\,1} & \delta_{3\,2} & \delta_{3\,3} & \ldots & \delta_{3\,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \delta_{n\,1} & \delta_{n\,2} & \delta_{n\,3} & \ldots & \delta_{n\,n} \end{bmatrix} \ .$$

In a dependency matrix, each row i is the dependency vector for the state interval of process i included in this system state. Since for all i, component i of process i's dependency vector records the index of process i's current state interval, the diagonal of the dependency matrix records the current state interval index of each process contained in that system state. These process states need not all have existed in the system at the same time; a system state is said to have *occurred* if all component process state intervals have each individually occurred.

A system state is called *consistent* if and only if it *could* have been seen at some instant by an outside observer during the preceding execution of the system, regardless of the relative speeds of the component processes [Chandy85]. In terms of the dependency matrix representing a system state, that state is consistent if and only if no value in any column is larger than the value in that column on the diagonal, indicating that no process depends on a state interval of another process after that process's own current state interval. That is, a system state $\mathbf{D} = [\delta_{**}]$ is consistent if and only if

$$\forall\, i, j\, [\delta_{j\,i} \leq \delta_{i\,i}] \ .$$

A system state is called *recoverable* if and only if it is consistent and each component process state interval is stable.

The set of system states that have occurred during any single execution of a system forms a lattice, called the *system history lattice*. In the lattice, a system state $\mathbf{A} = [\alpha_{**}]$ precedes another system state $\mathbf{B} = [\beta_{**}]$, written $\mathbf{A} \prec \mathbf{B}$, if and only if $\mathbf{A}$ *must* have occurred first during this execution. That is,

$$\mathbf{A} \preceq \mathbf{B} \iff \forall\, i\, [\alpha_{i\,i} \leq \beta_{i\,i}] \ ,$$

and

$$\mathbf{A} \prec \mathbf{B} \iff (\mathbf{A} \preceq \mathbf{B}) \wedge (\mathbf{A} \neq \mathbf{B}) \ .$$

For any two system states $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ that have occurred during the same execution, the *join* of $\mathbf{A}$ and $\mathbf{B}$ in this lattice, written $\mathbf{A} \sqcup \mathbf{B}$, represents a system state that has also occurred during this execution of the system, in which each process has received only those messages that it has received in *either* $\mathbf{A}$ or $\mathbf{B}$. That is, $\mathbf{A} \sqcup \mathbf{B} = [\theta_{**}]$, such that

$$\forall\, i \left[\theta_{i*} = \left\{ \begin{array}{ll} \alpha_{i*} & \text{if } \alpha_{i\,i} \geq \beta_{i\,i} \\ \beta_{i*} & \text{otherwise} \end{array} \right. \right] \ .$$

This construction copies each row from the corresponding row of either $\mathbf{A}$ or $\mathbf{B}$, depending on which dependency matrix has the larger value on its diagonal in that row.

The sets of consistent and recoverable system states that have occurred during this execution form sublattices of the system history lattice. Thus, during execution, there is always a *unique* maximum recoverable system state, which is simply the join of all elements in the recoverable sublattice. Logged messages and checkpoints are saved on stable storage until no longer needed for any possible future recovery of the system, and thus the current maximum recoverable system state never decreases.

At any time, the state to which the system will be restored if any failure were to occur at that time is called the *current recovery state*, and is always the unique maximum system state that is currently recoverable. After a failure, any process currently executing in a state interval beyond the state interval of that process in the current recovery state is called an *orphan*, since it depends on some state interval of another process that is not consistent with the current recovery state. In recovering the system to the current recovery state, each orphan is rolled back by forcing it to fail and recovering it in the same way as other failed processes.

4 Committable State Intervals

As described, the model defines a global view of the system. The definition of a consistent system state requires knowledge of the dependencies of all processes on one another, and the definition of a recoverable system state further requires all component process state intervals to be stable, requiring knowledge of the message logging and checkpointing progress of all processes in the system. As a basis for output-driven message logging and checkpointing, we extend the model to include a *local* view of the system, with the definition of a *committable* process state interval.

> **Definition 4.1** A state interval σ of some process i is *committable* if and only if there exists *some* recoverable system state in which process i is in any state interval $\rho \geq \sigma$.

This definition does not require any process to know any details of this recoverable system state, such as its dependency matrix or the state interval index of any process in it. When process i learns that such a recoverable system state exists, process i is said to *commit* state interval σ. By transitivity, for any state interval $\delta < \sigma$, state interval δ of process i is also committable if state interval σ is committable.

Logged messages and checkpoints must be saved on stable storage until it is guaranteed that they will not be needed to recover from any possible future failure in the system. Thus, since the set of recoverable system states that have occurred during any single execution of a system forms a sublattice of the system history lattice, once a process state interval becomes committable during an execution of the system, it remains committable for the remainder of that execution. Given the definition of a committable process state interval, the following lemma establishes sufficient conditions for a process removing its own logged messages and checkpoints from stable storage.

> **Lemma 4.1** If some state interval σ of a process i is committable, and if the effective checkpoint for state interval σ of process i 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 stated some state interval $\beta \leq \epsilon$
>
> cannot be needed for recovery from any possible future failure in the system and may be removed from stable storage.

Proof From Definition 4.1 and since the set of recoverable system states form a sublattice of the system history lattice, the state interval index of process i in the current recovery state cannot be less than σ for the remainder of this execution of the system. No checkpoint recording the state of process i in any state interval $\alpha < \epsilon$ can be needed for recovery from any possible future failure, since the checkpoint in state interval ϵ (or some later checkpoint) can be used to recover the state of process i. Likewise, no logged message received by process i to start some state interval $\beta \leq \epsilon$ can be needed for recovery from any possible future failure, since only messages received to start state intervals after the interval recorded in this checkpoint are needed. Thus, these checkpoints and logged messages may safely be removed from stable storage. □

The definition of a committable process state interval also allows a simple test to determine when output may be committed to the outside world. Output must be delayed until the fault-tolerance support can guarantee that no future failure in the system can force the sending process to roll back to a state interval before the output was sent. The following lemma establishes sufficient conditions for a process committing its own output to the outside world.

> **Lemma 4.2** If some state interval σ of a process i is committable, then any message sent to the outside world by process i from some state interval $\delta \leq \sigma$ may be committed.

Proof From Definition 4.1 and since the set of recoverable system states form a sublattice of the system history lattice, the state interval index of process i in the current recovery state cannot be less than σ for the remainder of this execution of the system. Output sent from any state interval $\delta \leq \sigma$ may be committed, since process i cannot be forced to roll back to any state interval before σ by any possible future failure. □

5 Output-Driven Message Logging and Checkpointing

5.1 Overview

With output-driven message logging and checkpointing, each process commits its own state intervals when necessary in order to allow its own output to the outside world to be committed. When a process sends output destined to the outside world, the process becomes responsible for first committing the state interval from which that output was sent. This commitment of process state intervals is performed using an efficient distributed algorithm initiated by the process whose state interval is being committed. Processes may also periodically create new checkpoints of their own current state in order to reduce the amount of potential reexecution necessary to restore their own state during recovery. This checkpointing does not affect the recovery of any other process, and need not be coordinated with any process's execution. The algorithm to commit a state interval may also be invoked by a process after recording a new checkpoint, in order to determine if messages received by the process before the checkpoint that have not yet been logged can be discarded, saving the overhead of recording them on stable storage. When a process commits a state interval, the process removes from stable storage any of its own logged messages and checkpoints that are no longer needed. Between invocations of the algorithm, processes cooperate with one another by piggybacking a small constant amount of information on each message sent by the underlying system.

This design concentrates on the use of message logging and checkpointing in allowing pending output to the outside world to be committed. In any system using message logging and checkpointing, only this use *forces* a process to log messages or to record new checkpoints during failure-free execution. Likewise, only this use forces commitment of any process state interval, or forces determination of the current recovery state or of any other recoverable system state during failure-free execution. The use of logged messages and checkpoints during failure recovery is less important if failures in the system are infrequent. Processes must not be forced to roll back "too far" if a failure

should occur, but this requirement is less urgent than output commitment and can be tuned to balance the possible amount of rollback against the message logging and checkpointing overhead of achieving this limit. Computation of the current recovery state in order to begin recovery after a failure is also less important if failures are infrequent. Furthermore, knowledge of the current recovery state or the commitment of process state intervals in order to determine which older process checkpoints and logged messages can be removed from stable storage is only an optimization to reduce the amount of stable storage space needed to support fault tolerance.

The algorithm used in output-driven message logging and checkpointing for committing a process state interval is based on the result of the following two lemmas.

> **Lemma 5.1** A stable state interval σ of some process i, with dependency vector $\langle \delta_* \rangle$, is committable if and only if for all $j \neq i$, state interval δ_j of process j is committable.

Proof *(If)* If for all j, state interval δ_j of process j is committable, there must exist some recoverable system state $\mathbf{A}_j$ in which process j is in some state interval $\alpha_j \geq \delta_j$. Let $\mathbf{R} = [\rho_{**}]$ be the join of of all $\mathbf{A}_j$ in the system history lattice. Since the set of recoverable system states form a sublattice of the system history lattice, $\mathbf{R}$ must also be recoverable. If $\rho_{ii} \geq \sigma$, then state interval σ is committable by the existence of $\mathbf{R}$. Instead, if $\rho_{ii} < \sigma$, consider the system state $\mathbf{R}' = [\rho'_{**}]$, where $\rho'_{i*} = \delta_*$, and for all $k \neq i$, $\rho'_{k*} = \rho_{k*}$. That is, $\mathbf{R}'$ is identical to $\mathbf{R}$, except that process i is in state interval $\sigma = \delta_i$. Since $\mathbf{R}$ is consistent, $\mathbf{R}'$ must be consistent. In column i of the dependency matrix of $\mathbf{R}'$, no value is larger than σ, the value in that column on the diagonal, since $\sigma > \rho_{ii}$. For each column $k \neq i$, no value is larger than $\rho'_{kk} = \rho_{kk}$, the value in that column on the diagonal, since by the definition of the join in the system history lattice, $\rho_{kk} \geq \alpha_k$, and by construction, $\alpha_k \geq \delta_k = \rho'_{ik}$. Therefore, $\mathbf{R}'$ is consistent. Since state interval σ is stable, $\mathbf{R}'$ is also recoverable, and thus, state interval σ is committable by the existence of $\mathbf{R}'$.

(Only if) By contradiction. Suppose state interval δ_k of some process k, $k \neq i$, is not committable, but state interval σ of process i is committable. Let $\beta < \delta_k$ be the maximum index of any state interval of process k that is committable. Since state interval σ is committable, there must exist some recoverable system state $\mathbf{R} = [\rho_{**}]$ in which process i is in some state interval $\rho_{ii} \geq \sigma$, by Definition 4.1. Since no component of a dependency vector can decrease through the execution of the process, component k of the dependency vector of state interval ρ_{ii} of process i, ρ_{ik}, must be at least as large as δ_k. Since $\mathbf{R}$ is consistent, the state interval index of process k in $\mathbf{R}$, ρ_{kk}, must be at least as large as ρ_{ik}. Therefore, in the recoverable system state $\mathbf{R}$, process k is in some state interval $\rho_{kk} \geq \rho_{ik} \geq \delta_k$. However, by the assumption, $\beta < \delta_k$ is the maximum index of any state interval of process k contained in an existing recoverable system state, leading to the required contradiction. □

> **Lemma 5.2** A state interval σ of some process i that is *not* stable is committable if and only if state interval $\sigma' > \sigma$ of process i is committable, where state interval σ' is stable and there does not exist another stable state interval δ such that $\sigma' > \delta > \sigma$.

Proof Follows from Definition 4.1 and from the fact that no component of the dependency vector of a process can decrease through the execution of the process. Choosing the minimum stable state

interval $\sigma' > \sigma$ of process i also minimizes all components of process i's dependency vector for the chosen state interval. Thus no later state interval of process i can be committable if state interval σ' is not also committable. □

5.2 Data Structures

Each process maintains its own current *state interval index* and *dependency vector*, as described in Section 3. Each process also maintains a *commit vector*, indexed by the identification of each process in the system. In the commit vector of each process i, component i records the maximum index of any state interval of process i that is committed. Each other component j, $j \neq i$, records the maximum index of any state interval of process j that process i knows is committable. When a process i is created, its dependency vector $\langle \delta_* \rangle$ is initialized to indicate that this process has no dependencies on any other process. The state interval index of the process, δ_i, is initialized to 0, and all other components of its dependency vector, δ_j for all $j \neq i$, are initialized to $\perp$. The commit vector $\langle \mu_* \rangle$ of a new process is initialized with all entries set to 0, since state interval 0 of each process is always committable due to the initial checkpoint of each process.

All messages received by a process are saved in a *message buffer* in the local volatile memory of the receiver. Messages are saved in this buffer until they are logged by writing the contents of the buffer to the message log on stable storage. This writing occurs asynchronously and does not delay the execution of the process. Normally, these messages are logged when the buffer is full. However, the state interval commitment algorithm may force some messages to be logged in order to commit a state interval, and messages in this buffer may instead be discarded and never written to stable storage if logging them becomes unnecessary first, based on Lemma 4.1.

As processes communicate, they cooperate to maintain their dependency and commit vectors. When some process i sends a message, it tags the message with

- its own current state interval index (δ_i in its own dependency vector), and
- its own current commit vector entry for itself, μ_i, giving the index of its latest committed state interval.

When some process j receives a message from process i, process j

- increments its own current state interval index (δ_j in its own dependency vector),
- sets the entry for process i in its own dependency vector, δ_i, to the maximum of its current value and the state interval index tagging this message, and
- sets the entry for process i in its own commit vector, μ_i, to the maximum of its current value and the commit vector entry tagging this message.

Components in the commit vector of a process may also be increased through the execution of the algorithm to commit a state interval.

5.3 The Algorithm

The algorithm to commit a process state interval is shown in Figure 1, using a remote procedure call notation. Each process uses the same algorithm. The function $CHECK\_COMMIT(\sigma, force)$ is executed locally to commit state interval σ of the local process. $NEED\_STABLE(\sigma, force)$ and $MAKE\_STABLE(\sigma)$ are executed by *CHECK_COMMIT* at other processes with which this process must cooperate to commit state interval σ. In calling *CHECK_COMMIT*, if *force* is **true**, the commitment of state interval σ is forced, by possibly causing other processes record new checkpoints or to log messages from their volatile message buffers. If *force* is **false** instead, *CHECK_COMMIT* simply checks if state interval σ can now be committed without forcing any new information to be recorded on stable storage.

The variable *pid* represents the identification of the local process. The dependency vector for each state interval δ of this process is represented by $DV(\delta)$, and the current commit vector of this process is represented by *COMMIT*. The new value for the commit vector is stored in the vector *NEW* until completion of the algorithm. The vector *NEED* records the maximum index of any state interval of each other process "needed" by the algorithm, and the vector *STABLE* records whether the corresponding state interval in *NEW* is known to be stable. The vector *UPDATE* is used as a parameter in communication with the function *NEED_STABLE*. Sparse representations of each vector are possible, in which elements of each $DV(\delta)$ and of *COMMIT*, *NEED*, *NEW*, and *UPDATE* are represented only if they are not equal to $\perp$, and elements of *STABLE* are represented only if the corresponding element of *NEW* is represented.

The algorithm essentially traverses the dependencies of state interval σ backwards, until each state interval being considered in the traversal is known to be committable. Since a recoverable system state contains only stable process state intervals, only state intervals that are stable or can be made stable are included in the traversal. For each state interval σ considered, *NEED_STABLE* chooses the minimum state interval index $\alpha \geq \sigma$ that meets these requirements. By choosing the minimum such α, all components of $DV(\alpha)$ are also minimized, since no component of the dependency vector can decrease through the execution of the process. Since only the maximum state interval of each process on which this process depends need be stable, the algorithm collects responses from each process in turn and sends out new *NEED_STABLE* requests only to the maximum state interval then needed. The test in function *NEED_STABLE* as to whether a state interval can be made stable normally returns "not found" only if the process has failed and the required messages for logging and the current state of the process have been lost. The extensions to the algorithm described in Section 5.6 add new interpretations to this test, though.

Definition 5.1 A commit vector $\langle \mu_* \rangle$ is *valid* if and only if for all i, state interval μ_i of process i is committable.

Lemma 5.3 At any time during the execution of a system using this algorithm, the commit vector of each process in the system is valid.

Proof By induction on the execution of the system. Let an event be any process changing its commit vector, and consider any total ordering of these events consistent with the partial ordering

```
function CHECK_COMMIT(σ, force)
    if σ ≤ COMMIT[pid] then return true
    NEW ← COMMIT;
    for j ← 1 to n do
        if DV(σ)[j] > COMMIT[j] then NEED[j] ← DV(σ)[j];
                                else NEED[j] ← ⊥;
    NEED[pid] ← ⊥;
    NEW[pid] ← σ;
    for j ← 1 to n do STABLE[j] ← true;
    while ∃ i such that NEED[i] ≠ ⊥ do
        (status, UPDATE) ← call NEED_STABLE(NEED[i], force) at process i;
        if status = "not found" then return false;
        if status = "committed" then
            for j ← 1 to n do
                COMMIT[j] ← max(COMMIT[j], UPDATE[j]);
                if NEED[j] ≤ COMMIT[j] then
                    NEED[j] ← ⊥;
                    NEW[j] ← COMMIT[j];
            if σ ≤ COMMIT[pid] then return true;
        else
            for j ← 1 to n do NEED[j] ← max(NEED[j], UPDATE[j]);
            NEED[i] ← ⊥;
            NEW[i] ← UPDATE[i];
            if status = "stable" then STABLE[i] ← true;
                                 else STABLE[i] ← false;
    for j ← 1 to n do
        if ¬STABLE[j] then call MAKE_STABLE(NEW[j]) at process j;
    COMMIT ← NEW;
    return true;

function NEED_STABLE(σ, force)
    if σ ≤ COMMIT[pid] then return ("committed", COMMIT);
    if force then α ← minimum sate interval index such that α ≥ σ and
                      state interval α of this process is stable or can be made stable;
         else α ← minimum sate interval index such that α ≥ σ and
                      state interval α of this process is stable;
    if no such state interval α exists then return ("not found");
    for j ← 1 to n do
        if DV(α)[j] > COMMIT[j] then UPDATE[j] ← DV(α)[j];
                                else UPDATE[j] ← ⊥;
    if state interval α is stable then status ← "stable";
                                  else status ← "volatile";
    return (status, UPDATE);

procedure MAKE_STABLE(σ)
    make state interval σ stable, either by logging messages or by checkpointing;
    return;
```

Figure 1 The algorithm to commit a process state interval

of events in the system imposed by the communication between processes [Lamport78]. Any two changes of commit vectors that are not ordered with respect to each other by this partial order cannot causally effect each other, and thus may appear in either order in the total ordering.

When a process is created, all components of its commit vector are initialized to 0. Since state interval 0 of each process is stable from its initial checkpoint, this initial commit vector is valid.

When some process k receives a message from some process j, process k sets the entry in its commit vector for process j to the maximum of its current value and the commit vector entry tagging this message. Let δ be the commit vector entry tagging the message. When the message was sent, state interval δ of process j was committable. Since checkpoints and logged messages are not removed from stable storage until no longer needed (Lemma 4.1), this state interval is still committable when process k receives the message and modifies its own commit vector. Thus, process k's commit vector remains valid.

During the execution of the function *CHECK_COMMIT* by some process k, if some call to *NEED_STABLE* at a process i returns "committed", process k replaces each component of its commit vector with the maximum of its current value and the corresponding component of *UPDATE*. Since the returned value of *UPDATE* is a copy of the commit vector of process i, process k's commit vector remains valid by the argument above, applied to each component j.

When the **while** loop of function *CHECK_COMMIT* terminates, process k replaces each component of its commit vector with the corresponding component of *NEW*. The loop tests if state interval σ of the local process is committable, by applying Lemma 5.1. The vector *NEED* is maintained such that for all i, if $NEED[i] \neq \perp$, then state interval $NEED[i]$ of process i has not yet been tested for being committable. Since the definition of a committable state interval is transitive, only the maximum index of any state interval of each process that must be tested for being committable is saved in *NEED*. Process k also maintains the vector *NEW* such that each $NEW[j]$ records the maximum index of any state interval of process j that it knows will also be committable if state interval σ is committable. Process k also maintains the vector *STABLE* such that each $STABLE[j]$ = **true** if and only if state interval $NEED[j]$ of process j is known to be stable. The **while** loop terminates when all $NEED[i] = \perp$, implying that all necessary state intervals have been tested for being committable, according to Lemma 5.1. Process k then ensures that each state interval $NEW[j]$ is stable, using the value of $STABLE[j]$ maintained in the loop. Once all necessary *MAKE_STABLE* calls have completed, *NEW* is then a valid commit vector and is copied to process k's commit vector. □

Theorem 5.1 The function *CHECK_COMMIT*, executed by some process k for some stable state interval σ, returns **true** if and only if state interval σ of process k is committable.

Proof Follows directly from Lemma 5.3 and Definition 4.1. □

The algorithm is shown in Figure 1 using synchronous remote procedure calls, but the performance of the algorithm can easily be improved using asynchronous remote procedure calls and by the use of unreliable broadcast if provided by the network hardware. In the **while** loop of

function *CHECK_COMMIT*, the *NEED_STABLE* requests to individual processes can be sent asynchronously, with multiple requests outstanding. The results are then collected at the bottom of the loop, and are used to evaluate the loop predicate for the next iteration. If available, a single unreliable broadcast of the *NEED* vector can be used instead on each iteration to effectively send the *NEED_STABLE* request to all processes, such that each receiving process ignores the request if its corresponding entry in the vector received is $\perp$. Any *NEED_STABLE* requests lost by the unreliable broadcast will effectively be retransmitted on the next iteration of the loop. Likewise, the *MAKE_STABLE* requests sent by *CHECK_COMMIT* can be sent either with a single round of asynchronous requests to all necessary processes or with an unreliable broadcast that is retransmitted until all replies are received. This optimization also enables all processes that must make new state intervals stable in order to complete the execution of *CHECK_COMMIT* to do so in parallel.

The performance of the algorithm can also be improved by notifying other processes when commit vector entries are increased by an execution of *CHECK_COMMIT*, since only dependency vector entries $DV(\delta)[j] > COMMIT[j]$ must be tested by the algorithm for being committable. Tagging each message sent with the current commit vector entry of the sender allows the receiving process to inexpensively maintain recent information in its own commit vector about the processes from which it receives messages, which are the processes on which it depends. However, the execution of *CHECK_COMMIT* by some process k may also increase the value of process k's commit vector entries for processes other than process k. Before copying *NEW* to *COMMIT* at the completion of *CHECK_COMMIT*, process k can send the new commit vector entry $NEW[j]$ to each process j for which $NEW[j] \neq COMMIT[j]$. Each process j then sets its own commit vector entry for itself to the maximum of its current value and the value received from process k. If done with a reliable remote procedure call, this ensures that each process knows its maximum committable state interval that has been discovered by any process. If unreliable broadcast is available on the network, process k can alternatively broadcast the new value of *COMMIT* to all processes in the system, which then set each component in their own commit vectors to the maximum of its current value and the value received in this broadcast. Reliable transmission of this broadcast is not required, since the algorithm works correctly for any valid commit vector at each process.

5.4 Usage

When a process sends output to the outside world from some state interval σ, the output is saved in a buffer in memory, and the process invokes

$$CHECK\_COMMIT(\sigma, \textbf{true})$$

before committing the output from this buffer. This forces state interval σ of this process to be committed. This may also cause other processes on which this state interval depends to log messages to stable storage, but these messages would normally be logged eventually with other optimistic message logging algorithms [Strom85, Johnson88, Sistla89] as well. The process may continue normal execution in parallel with the execution of *CHECK_COMMIT*, but the output must be held until *CHECK_COMMIT* completes.

When the process creates a new checkpoint recording some state interval σ, it may also invoke

$CHECK\_COMMIT(\sigma, \textbf{false})$

to determine if the state interval recorded in the checkpoint is committable. If so, the messages received by the process before the checkpoint that are still in the volatile message buffer of the process can be discarded, without being logged on stable storage. Otherwise, the process may log these messages to free space in the message buffer, but it is not required to do so. Alternatively, the process may use *CHECK_COMMIT* after creating a new checkpoint, to enable it to remove older checkpoints after this new one has been recorded. Suppose that the process is willing to retain only some maximum number of checkpoints, c, and let ω be the index of the state interval of the process that is recorded in the oldest checkpoint that the process is still willing to retain after recording this new checkpoint. Then, the process can invoke

$CHECK\_COMMIT(\omega, \textbf{true})$

to enable it to then remove all checkpoints recording state intervals less than ω, since state interval ω will then be committable. By using this procedure only after each d new checkpoints of the process, the maximum number of checkpoints that must be retained by the process can inexpensively be bounded between c and $c+d$. Such decisions on checkpoint maintenance are entirely local to each process.

5.5 Failure Recovery

After a failure, each surviving process is checkpointed and all messages in the message buffer of each surviving process are logged. This allows the recovery procedure to be restartable and ensures that no additional work in the system can be lost if further failures occur during recovery. The recovery procedure then uses only information that has been recorded on stable storage.

To begin recovery, each failed process collects the current commit vector from all surviving processes, and sets each component in its own commit vector to the maximum of the corresponding components in the commit vectors received. The process is then recovered to its most recent stable state interval. Suppose for some process, this state interval is σ, with an effective checkpoint recording state interval ϵ of the process. The process is reloaded from this checkpoint to state interval ϵ, and the process then reexecutes using logged messages to reach state interval σ. The necessary effective checkpoint and logged messages must be available on stable storage, since checkpoints and logged messages are not removed from stable storage until no longer needed (Lemma 4.1).

Orphan processes that result from the failure are eliminated using an *incarnation number* scheme similar to that used in Strom and Yemini's Optimistic Recovery system [Strom85]. Each process maintains an incarnation number recording the number of times that the process has rolled back. When a recovering process begins a new state interval after completing reexecution to its most recent stable state interval, it begins a new incarnation as well. When the process increments its state interval index for the new state interval, it also increments its incarnation number. Any surviving process that has become an orphan due to the failure will be forced to roll back to a state interval before it received the message making it an orphan. The orphan process is recovered using

its effective checkpoint for this state interval, and begins a new incarnation when it begins a new state interval after the required reexecution using its logged messages. The state interval index of a process is always tagged with its current incarnation number. All state interval indices tagging messages or used by the commitment algorithm are tagged with the corresponding incarnation number of that process.

When a process begins a new incarnation, it broadcasts its new incarnation number and state interval index to all processes in the system. If any process depends on a later state interval of the failed process, this process has become an orphan due to the failure. If the broadcast is reliable, then all orphan processes are reliably detected. Otherwise, any remaining orphan processes are detected during normal execution of the system when two processes communicate, either through a message between them from the underlying system or through a message from the commitment algorithm. If a received message was sent from a state interval of the sender that the receiving process knows has been rolled back (from the incarnation numbers), the receiver determines that this sender is an orphan. Likewise, if the message is a *NEED_STABLE* request from the algorithm and indicates that the sender depends on a state interval and incarnation of the receiver later than the receiver's current incarnation, the receiver determines that it is currently an orphan itself. In this case, the sending process is also an orphan, since it depends on this state interval of the receiver. Whereas Strom and Yemini's incarnation number scheme assumes reliable notification that eventually reaches all processes when a process begins a new incarnation, our scheme allows simpler unreliable broadcast that may occasionally lose some notifications. We are currently developing some aspects of this scheme, but its details do not affect the algorithm as presented in this paper.

This paper does not address the problem failure detection within the system. However, the execution of the commitment algorithm must be coordinated with the specific failure detection method used, such that any remote procedure call by the algorithm that fails due to the failure of the target process will result in the detection of that failure. In this case, the execution of the algorithm is terminated and the recovery of that failure is initiated. For example, if process k is executing the algorithm and some *MAKE_STABLE* call to process i detects the failure of process i, the current *CHECK_COMMIT* call at process k is terminated. This *MAKE_STABLE* call by process k indicates that process k depends on a volatile state interval of process i that was lost when process i failed. Process k thus becomes an orphan process due to the failure, and must be rolled back to some state interval before this dependency on process i was created. After the recovery has been completed, the terminated call of *CHECK_COMMIT* is no longer needed, since process k has been rolled back to a state interval before the one that was being committed.

5.6 Extensions

When a process receives a *NEED_STABLE* request and the needed state interval σ is not currently stable, a great deal of flexibility exists in the particular state interval α that the process chooses, while still maintaining the correctness of the algorithm. In particular, the process can decide to use *any* state interval $\alpha \geq \sigma$ that is currently stable or that it can make stable. For example, the process may decide to create a new checkpoint of its current state interval for this α, if its previous checkpoint was made some time ago. This helps to limit the amount of reexecution necessary for

any individual process to be restored to a particular state interval during recovery, and may avoid logging the messages received before this new checkpoint.

This extension can be carried further, as each process can independently decide not to log *any* messages during arbitrary periods of its own execution. During such a period, any *NEED_STABLE* request must be satisfied by creating a new checkpoint of the process, but this allows the process to avoid all overhead of message logging. Such a period of not logging messages must be terminated by some future checkpoint of the process. This extension may be particularly useful for a process that receives many messages but sends few messages, such that it would otherwise have many messages to log but few other processes would have dependencies on different state intervals of the process that could cause future *NEED_STABLE* requests to the process. If all processes in the system decide to use this option, the commitment algorithm becomes an efficient global checkpointing algorithm [Koo87, Chandy85], in which the minimum number of processes are forced to record a new checkpoint to commit any state interval.

Finally, this extension can be used to easily support arbitrary periods of nondeterministic execution in any process, provided that the process can detect when its own execution is nondeterministic. Once such nondeterminism is detected, the process simply enters a period of not logging messages as described above, which may be ended on any future checkpoint of the process. If the process again detects nondeterministic execution after it has started logging new messages after this checkpoint, the process simply enters a new period of not logging messages. For any *NEED_STABLE* request to the process during such a period, the process creates a new checkpoint to create the needed stable state interval. For each message sent during such a period, the process also first increments its own state interval index and tags the message with this new value. Since the receiver records only the maximum index of any state interval of this sender on which it depends, this allows the receiver to differentiate between the individual state from which this message was sent and any earlier execution of the same process that might otherwise be in the same in the same state interval.

6 Related Work

Previous algorithms using optimistic message logging methods [Strom85, Johnson88, Sistla89] have attempted to maintain current knowledge of the maximum possible recoverable system state, so that output can be committed from the system without excessive delays. Our algorithm takes the alternative approach of determining recoverable system states only when necessary for output commitment, and requires only the minimum number of other processes to cooperate in this commitment. Our previous work with optimistic message logging included a centralized incremental algorithm for finding the current recovery state [Johnson88, Johnson89], and the algorithm presented here is derived in part by distributing the execution of this centralized algorithm. These systems are unique in including checkpointed process state intervals in determining the existence of recoverable system states, and it is this feature that allows us to easily integrate support for optionally using checkpointing alone and for nondeterministic process execution. Our support for global checkpointing is similar to Koo and Toueg's algorithm [Koo87], in that we require only a

minimum number of other processes to record new checkpoints, but we also allow processes individually to instead log messages to make a state interval stable, rather than requiring a complete new checkpoint. Also, the output-driven approach of our algorithm allows us to only initiate a new global checkpoint (or any recoverable system state) when it is required in order to commit output from the system.

Our goal of not logging some messages on stable storage is similar to the work of Strom, Bacon, and Yemini [Strom88], but they use a different approach. Their algorithm saves a copy of each message in the volatile memory of the *sender* until it is no longer needed for recovery, but must then return a *receive sequence number* to indicate the order of receipt of each message to the sender. Also, they must eventually "spool" these messages from volatile memory to stable storage in order to be able to remove old checkpoints from stable storage. Instead, our algorithm saves messages in the volatile memory of the *receiver* and thus requires no receive sequence number, and is able to remove old checkpoints and discard these messages from volatile memory without writing them to stable storage.

The approach of output-driven message logging and checkpointing is also similar to the use of stable storage to support atomic transactions and atomic actions [Oki85, Spector87, Haskin88]. Recovery information needed by the transaction may be written asynchronously to stable storage during the execution of the transaction, but need only be forced to stable storage when the transaction commits. This is similar to the need for state intervals to be stable in our commitment algorithm. However, we allow many of the messages that would otherwise be part of the recovery information on stable storage to be discarded before logging. Also, since the use of message logging and checkpointing is transparent, applications need not be structured as transactions, and we do not require the notion of an explicit transaction commit.

7 Conclusion

Output-driven optimistic message logging and checkpointing efficiently supports transparent fault tolerance in distributed systems. All output to the outside world is committed quickly after being sent, while reducing the overhead required for output commitment by previous optimistic message logging methods [Strom85, Johnson88, Sistla89]. Each process commits its own output, with the cooperation of the minimum number of other processes, allowing the method to scale to large systems with many processes. Processes may execute either deterministically or nondeterministically. Only the need to commit output forces any process to log received messages or to checkpoint. Any message not needed to allow pending output to be committed need not be logged. Individual processes may also dynamically decide to use checkpointing without message logging, to avoid the expense of logging a large number of messages or to support their own nondeterministic execution.

We are currently implementing output-driven optimistic message logging and checkpointing in a system of SUN workstations running the V-System [Cheriton83, Cheriton88] on an Ethernet network. The algorithm allows each process a great deal of freedom in choosing between using message logging and checkpointing or using checkpointing alone, and in the use of the algorithm to reduce the number of messages that must be logged or to limit the number of checkpoints that

must be retained, as described in Section 5.6. We are working on appropriate policies to enable each process to effectively make these choices to reduce its own fault-tolerance overhead.

References

[Borg83] 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.

[Borg89] 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.

[Chandy85] 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.

[Cheriton83] David R. Cheriton and Willy Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 129–140. ACM, October 1983.

[Cheriton88] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[Haskin88] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988.

[Johnson87] 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.

[Johnson88] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181. ACM, August 1988.

[Johnson89] David B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. Ph.D. thesis, Rice University, Houston, Texas, December 1989. Also available as Technical Report Rice COMP TR89-101, Department of Computer Science, Rice University, December 1989.

[Koo87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[Lamport78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Oki85] Brian M. Oki, Barbara H. Liskov, and Robert W. Scheifler. Reliable object storage to support atomic actions. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 147–159. ACM, December 1985.

[Powell83] 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.

[Schlichting83] 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.

[Sistla89] 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*. ACM, August 1989.

[Spector87] Alfred Z. Spector. Distributed transaction processing and the Camelot system. In *Distributed Operating Systems: Theory and Practice*, edited by Yakup Paker, Jean-Pierre Banatre, and Müslim Bozyiğit, volume 28 of *NATO Advanced Science Institute Series F: Computer and Systems Sciences*, pages 331–353. Springer-Verlag, Berlin, 1987. Also available as Technical Report CMU-CS-87-100, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, January 1987.

[Strom85] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[Strom88] 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.