On the Use and Implementation of Message Logging

Elmootazbellah N. Elnozahy
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Willy Zwaenepoel
Department of Computer Science
Rice University
Houston, TX 77251

Abstract

Message logging has long been advocated as offering better failure-free performance than coordinated checkpointing. On the contrary, we present a number of experiments showing that for compute-intensive applications executing in parallel on clusters of workstations, message logging has higher failure-free overhead than coordinated checkpointing. Message logging protocols, however, result in much shorter output latency than coordinated checkpointing. Therefore, message logging should be used for applications involving substantial interactions with the outside world, while coordinated checkpointing should be used otherwise.

We also present an unorthodox message logging design that uses coordinated checkpointing with message logging, departing from the conventional approaches that use independent checkpointing. This combination of message logging and coordinated checkpointing offers several advantages, including improved failure-free performance, bounded recovery time, simplified garbage collection, and reduced complexity. Meanwhile, the new protocols retain the advantages of the conventional message logging protocols with respect to output commit.

Finally, we discuss three "lessons learned" from an implementation of various message logging protocols. First, during output commit, only the dependency information for the messages in the log needs to be written to the stable storage. It is not necessary to write the message data to stable storage, leading to faster output commit. Second, the use of copy-on-write in the implementation of message logging substantially reduces the logging overhead for communication-intensive programs. Finally, we provide quantitative evidence supporting previous qualitative claims about the superiority of sender-based message logging over receiver-based logging.

*This work was supported in part by NFS Grants CDA-9222911 and CCR-9116343, and by the Texas Advanced Technology Program Grants ATP 003604012 and ATP 0036041014. The first author was also supported in part by an IBM Graduate Fellowship, and by the Advanced Research Projects Agency under contract number DABT63-93-C-0054. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies of the sponsors.

1 Introduction

Many methods have been proposed for rollback-recovery based on message logging [12, 16–20, 33, 35, 36, 40, 41]. During failure-free operation, the processes participating in a distributed computation take independent checkpoints and log the messages that they exchange. When a failure occurs, a recovery algorithm uses the message logs and checkpoints available on stable storage to compute a consistent state [8] to which the processes roll back. Message logging allows the processes in a distributed computation to take independent checkpoints while avoiding the possibility of the domino effect [31] during recovery. It has been argued that such a design results in better failure-free performance than coordinated checkpointing [14, 23, 24, 27, 32, 34, 37] because it avoids the overhead of synchronizing the checkpoints to form a consistent state. This premise is true for environments where communication is expensive, because the incremental cost of logging then becomes small, while the message exchanges necessary to synchronize the checkpoints add substantial overhead (for example, see Barghava et al. [4]).

In workstation clusters, however, the cost of network communication is small and rapidly decreasing. Meanwhile, the cost of accessing stable remains high because of the mechanical nature of disks, which are the media of choice for implementing stable storage because of the cost per capacity advantage over other techniques. Recent experiments [11] have indicated that the difference in performance between coordinated and independent checkpointing becomes marginal in workstation clusters. This paper re-examines the design premises of message logging protocols in light of these developments.

We have implemented three message logging protocols and compared their performance to an implementation of coordinated checkpointing [11] on the same hardware and software platform. The protocols are i) receiver-based optimistic message logging [16, 33], ii) sender-based optimistic message logging [18, 35], and iii) the Manetho rollback-recovery system [10, 12]. All three protocols use asynchronous message logging [36]. Results show that all three message logging protocols perform *worse* than coordinated checkpointing. The costs of writing the message logs and managing the recovery information on stable storage outweigh the cost of coordinating the checkpoints during failure-free operation and the overhead of contention on the stable storage server during the global checkpoint.

This result does not, however, imply that coordinated checkpointing is always the method of choice. Message logging reduces the output latency during interactions with the outside world. In a rollback-recovery system, before a process is allowed to send output to the outside world, an output commit algorithm must guarantee that the state from which the application is producing output will not be rolled back because of a failure. Output latency is the time necessary to execute the output commit algorithm. Our measurements show that message logging protocols commit output much faster than coordinated checkpointing.

Based on these observations, we present a new message logging protocol that uses *coordinated* rather than independent checkpointing. This unorthodox combination retains the fast output commit property of conventional message logging using independent checkpointing, but offers several advantages not present in conventional designs. First, the message logs need not be written to stable storage, resulting in a reduction in stable storage access and enhanced performance. Second, no explicit garbage collection algorithm is needed. Events that occurred before the last consistent checkpoint will never be rolled back and recovery information related to these events can be discarded as soon as a consistent checkpoint is completed. Third, the use of coordinated checkpointing guarantees that processes never roll back beyond their latest checkpoint. This new design thus offers a better bound on recovery time and requires only one permanent checkpoint per process to be maintained on stable storage. We have implemented this new approach, and measurements from this implementation indicate that it achieves better failure-free performance than message logging protocols using independent checkpointing.

Finally, we highlight three issues that would be of interest to future implementors of message logging systems. First, previous message logging designs write a part of the message log to stable storage during output, including both the message data in the log as well as the dependency information. We show that only the dependency tracking information needs to be saved on stable storage, resulting in reduced output latency. Second, the use of copy-on-write in implementing the message log can reduce the overhead of logging. Copy-on-write extends the message log into the application's address space, reducing the probability of overflowing the volatile log and the resulting potential for blocking the application. Finally, our results show that sender-based logging has failure-free performance superior to receiver-based logging. While qualitative arguments for sender-based logging have been made before [18, 35], we are not aware of any quantitative performance comparisons that support these arguments.

The experiments presented here focus on the failure-free performance of the various protocols under study. We do not consider in this paper the performance when failures and recoveries occur. Also, our results only apply to processes where nondeterminism can be tracked efficiently, which is the underlying assumption of all message logging protocols.

The outline of this paper is as follows. Section 2 reviews some background information. Section 3 describes the environment used for the experiments. Section 4 presents the results of the experiments comparing conventional message logging protocols to coordinated checkpointing. Section 5 presents the new message logging protocols based on coordinated checkpointing. Finally, Section 6 compares our work with related research and Section 7 concludes the paper.

2 Background

This section summarizes the rollback-recovery techniques discussed in this paper. They include coordinated checkpointing, optimistic message logging with receiver-based and sender-based logging, and the Manetho rollback-recovery system. For a full description of these methods, we refer the reader to the relevant publications.

2.1 Coordinated Checkpointing

With coordinated checkpointing, processes cooperate so that their checkpoints form a consistent state [8]. Once a new global consistent state is recorded, the checkpoints belonging to the previous one may be discarded. As a result, no garbage collection of old checkpoints is required. During recovery, the application rolls back to the last consistent checkpoint, thereby bounding the amount of lost work. Drawbacks of coordinated checkpointing include the overhead required for the coordination protocol and the high load on the stable storage service during checkpointing. For this study, we used the algorithm described in our previous study of checkpointing in workstation clusters [11].

2.2 Receiver-Based Message Logging

With receiver-based message logging (RBML) [6, 19, 20, 30, 33, 36], the processes participating in a distributed computation log on stable storage the messages that they receive during failure-free operation. Asynchronous logging is used, since the failure-free overhead of synchronous logging is too expensive without hardware support [6, 30].

During recovery from a failure, a process restarts from a previous checkpoint and replays the messages in the log to restore the execution to a state that occurred before the failure. As with all message logging protocols, process execution must be deterministic in order for message replay to restore a process to the same state as before the failure. Several techniques exist for recovery, all based on computing the maximum recoverable state using the checkpoints and message logs available on stable storage [16, 33]. In all these techniques, the failure of one process may cause other processes to roll back as well, even if those processes survive the failure. These processes are called orphans [36]. Furthermore, a garbage collection protocol is required to reclaim old checkpoints and messages, and each process may have to maintain several checkpoints on stable storage. For our study, we use the technique suggested by Johnson and Zwaenepoel [19]. In this technique, the sender adds $O(1)$ dependency information on each message it sends. This information is used during recovery to compute the maximum recoverable state.

2.3 Sender-Based Message Logging

With RBML, when a process fails, the volatile message log is lost. In sender-based message logging (SBML), messages are logged in the sender's volatile storage [18]. If a process fails, the messages needed for execution replay are still available in their senders' logs. This technique tolerates a single failure in the system. Strom et al. enhanced this technique by separating the logging of the message data from its receipt order [35]. The data of a message is still logged at the sender, while its receipt order is logged at the receiver. The resulting protocol tolerates an arbitrary number of failures. Like RBML, SBML requires the maintenance of several checkpoints per process on stable storage, and a garbage collection protocol to reclaim old checkpoints and messages. In our implementation, each sender adds an $O(1)$ dependency information on each message it sends, as suggested by Johnson [16]. The recovery protocol uses the dependency information in computing the maximum recoverable state.

2.4 The Manetho System

Manetho uses a combination of independent checkpointing, sender-based message logging, and dependency tracking by means of an antecedence graph [10, 12]. The antecedence graph provides every process in the system with a complete history of the nondeterministic events that have causal effects on its state. Adding the antecedence graph to sender-based message logging results in several advantages not present in optimistic logging protocols. First, a process commits output locally without having to run a multihost protocol [12]. Second, a process cannot become an orphan as a result of a failure of another process. Third, no process rolls back beyond its most recent checkpoint, and each process needs to maintain only one permanent checkpoint on stable storage. The antecedence graph thus allows a message logging protocol to combine the advantages of pessimistic and optimistic logging without their disadvantages. The price paid is the cost of maintaining the antecedence graph during failure-free operation and a complex garbage collection algorithm to reclaim obsolete information in the antecedence graph. Several implementation techniques are used to reduce this cost [10].

3 Experimental Setup

We have implemented the four techniques described in Section 2 on the same software and hardware platform. In this section, we describe the experimental setup and the application programs used for this study.

3.1 Experimental Environment

The implementations were carried out on a dedicated 10 Mbit/sec Ethernet connecting 16 diskless Sun-3/60 workstations. Each workstation is equipped with a 20-MHz Motorola MC68020 processor and 4 megabytes of memory, of which 740 kilobytes are consumed by the operating system, and 512 Kilobytes are consumed by the message logs when applicable. These machines run a version of the V-System distributed operating system [9]. Stable storage is provided by two Sun-3/140 network file servers, each using a 16-MHz MC68020 processor and a Fujitsu Eagle disk.

3.2 Applications

We present performance measurements for four long-running, compute-intensive applications.* These applications are typical of those that would exploit the parallel processing capacity of a workstation cluster. All measurements were carried out with the applications executing on 16 machines. The network and workstations were otherwise idle. Table 1 summarizes the running times, the memory requirements and the communication rates for the four applications. A description follows:

- `gauss` performs Gaussian elimination with partial pivoting on a 1024×1024 matrix. In each iteration each process communicates with the process currently holding the current pivot element.
- `grid` performs an iterative computation on a grid of 2048×2048 points. In each iteration, the value of each point is computed as a function of its value in the last iteration and the values of its neighbors. The grid is subdivided among the processes participating in the computation. After each iteration, each process exchanges the values on the borders of its subgrid with the processes that contain the bordering subgrids. This application occurs in the kernel of many fluid-flow modeling algorithms.
- `sparse` solves a sparse system of linear equations in 48000 unknowns, using a variation on the iterative Gauss-Seidel method. The system is sparse in that less than 0.25% of each row in the matrix is nonzero. Each process is responsible for computing a portion of the solution vector. After each iteration, each process sends the computed subvector to the other processes.

*Readers familiar with our previous work may recall that we used eight applications in our evaluation of the performance of coordinated checkpointing [11]. The four applications not included here (`fft`, `matmult`, `nqueens`, and `prime`) have communication loads similar to `tsp` and lead to similar conclusions. We also use a smaller problem size for `sparse` because the space taken up by the message logs reduced the amount of memory available to the application.

Program Name	Running Time (minutes)	Per Process Memory (Kbytes)			Total Memory	Communication Rate (per second)	
		Code	Data	Total	Mbytes	Messages	Kbyte
gauss	48	20	576	596	9.5	49.5	31.7
grid	59	21	2163	2184	35.1	16.0	8.3
sparse	57	22	1954	1976	31.6	88.5	190.3
tsp	73	21	27	48	0.8	0.2	0.1

Table 1 Application running times, memory requirements, and communication rates.

- `tsp` uses a distributed branch-and-bound algorithm to solve the traveling salesman problem for a dense map of 18 cities. The program has a master-slave structure. A master maintains the current best solution, and a task queue containing subsets of the search space. All communication occurs between the slaves and the master.

4 Performance of Traditional Message Logging

4.1 Failure-Free Performance

We compared the failure-free overhead of coordinated checkpointing with that of the three message logging protocols described in Section 2. Table 2 shows the results of these experiments. The overhead is expressed by the percent increase in running time due to the provision of fault tolerance under the four protocols. For completeness, we also show the overhead for a protocol that uses independent checkpointing without message logging. Measurements are shown for checkpointing intervals of two, five, and ten minutes for each application. No garbage collection was performed in any of the message logging protocols, and therefore the measurements underestimate their actual overhead.

The measurements show that coordinated checkpointing outperforms the three message logging protocols, except for `tsp`, which has very little communication. To further illustrate this point, Table 3 shows the components of the overhead for each of the four protocols. The table shows the added overhead *beyond* the cost of independent checkpointing.[†] For coordinated checkpointing, this is the overhead of coordination. For both RBML and SBML, this is the overhead of logging the messages. The cost of dependency tracking for both protocols is negligible. For Manetho, the overhead is split between the overhead of message logging and dependency tracking. Except for `tsp`, the overhead due to coordinating the checkpoints is less than the combined overhead of logging the messages and tracking the dependencies.

We conjecture that our results will hold on other modern hardware platforms as well. Processor speed and network bandwidth are increasing, resulting in higher communication rates. Stable storage bandwidth is not improving at the same rate, increasing the relative cost of message logging. Stable storage bandwidth could be improved by distributing the load over multiple servers. We expect coordinated checkpointing to benefit more from such a stable storage organization, because most of the overhead of coordinated checkpointing results from all machines accessing stable storage at approximately the same time to record their checkpoints. No such contention was observed in saving the message logs.

4.2 Output Interactions

Table 4 shows the average latency of releasing output to the outside world according the output commit algorithms for the four rollback-recovery protocols under study in this section. The measurements are shown for the case where n processors are participating in the output commit algorithm. The latency is expressed as a function of the size of the global checkpoint in Megabytes for the coordinated checkpointing protocol. For RBML and SBML, the overhead is expressed as a function in the number of processors involved and the size of the data to be logged. For Manetho, output commit does not require a multihost protocol like the other systems, and therefore output commit has a constant overhead.

The table shows that the three message logging protocols have better latencies than the coordinated checkpointing protocol. The Manetho system has the lowest latency because its dependency tracking allows each process to commit output locally with a single stable storage I/O. The other two message logging protocols require a multihost protocol to commit output, which entails exchanging several messages and performing several stable storage I/O operations [16, 17, 33].

[†]The overhead does not always "add up" to the total value because of the overheads of the various components overlap during the execution. Also, for `gauss`, coordinated checkpointing is shown to have a negative additional cost compared to independent checkpointing. With short checkpointing intervals, coordinated checkpointing performs better than independent checkpointing with `gauss` because of the tight synchronization pattern of this application and its interaction with the coordination [11].

Program Name	% Increase in running time					
	Checkpointing Interval	Independent Checkpointing	Coordinated Checkpointing	SBML	RBML	Manetho
`gauss`	2 min.	0.8	0.3	1.0	2.0	1.3
	5 min.	0.3	0.2	0.5	1.3	1.2
	10 min.	0.1	0.1	0.2	1.1	1.1
`grid`	2 min.	1.6	1.8	1.9	2.3	2.8
	5 min	0.3	0.6	0.6	0.9	1.6
	10 min.	0.2	0.3	0.4	0.8	1.3
`sparse`	2 min.	1.6	2.0	4.1	9.8	5.6
	5 min	0.3	0.6	3.2	8.8	4.4
	10 min.	0.2	0.3	2.8	8.5	3.8
`tsp`	2 min.	0.0	0.0	0.0	0.0	0.0
	5 min	0.0	0.0	0.0	0.0	0.0
	10 min.	0.0	0.0	0.0	0.0	0.0

Table 2 Failure-free overhead of independent and coordinated checkpointing and three message logging protocols.

Program Name	% Increase in running time					
					Manetho	
	Interval	Coordination Overhead	SBML Logging	RBML Logging	Logging	Dependence
`gauss`	2 min.	-0.5	0.2	1.2	0.2	0.7
	5 min.	-0.1	0.2	1.1	0.2	0.7
	10 min.	0.0	0.2	1.1	0.2	0.7
`grid`	2 min.	0.2	0.3	0.6	0.3	0.6
	5 min	0.3	0.3	0.6	0.3	0.6
	10 min.	0.1	0.3	0.6	0.3	0.6
`sparse`	2 min.	0.2	2.5	8.2	2.5	0.9
	5 min	0.2	2.6	8.2	2.6	0.9
	10 min.	0.1	2.5	8.2	2.5	0.9
`tsp`	2 min.	0.0	0.0	0.0	0.0	0.0
	5 min	0.0	0.0	0.0	0.0	0.0
	10 min.	0.0	0.0	0.0	0.0	0.0

Table 3 Added overhead due to particular aspects of each protocol.

Output Commit Latency (Seconds)			
Coordinated Checkpointing	SBML	RBML	Manetho
1.210 + 1.3/MB	$0.020 + 0.038n$ + 1.3/MB	$0.020 + 0.038n$ + 1.3/MB	0.050

Table 4 The output latency for coordinated checkpointing and three protocols based on message logging.

4.3 "Lessons Learned"

4.3.1 Dependency Logging on Output Commit

Conventional message logging requires the message data to be flushed to stable storage before committing output [16, 33, 36]. By inspecting these algorithms, we found that for the purpose of committing output, it suffices to write the receipt order of the messages instead of the message data. Consider for instance traditional RBML. The information required to replay a non-orphan message during recovery consists of its data and receipt order. The message receipt order must be retrieved from the log, as it cannot be inferred otherwise. The message data, on the other hand, can be retrieved from the log, or it can be regenerated during recovery since the message is not an orphan [10]. This result suggests that for RBML and SBML, flushing the message data for committing output is not necessary; nor is it efficient, as shown by the measurements. By just flushing the dependency tracking information that consists only of the receipt orders, lower latency in committing output can be obtained. For both RBML and SBML, the latency would be only $20 + 38n$ milliseconds when n is the number of processes that have to participate in the output commit algorithm.

4.3.2 Copy-on-write and Message Logging

With asynchronous logging, process execution may still be blocked when the volatile log becomes full while logging a message. Much of this blocking can be avoided by using *copy-on-write* protection on the message data in the application address space. When the volatile log fills up, the memory management unit (MMU) information is modified to write-protect the pages containing the message data in the application address space, but the application continues to execute. As the volatile log is written to stable storage, and new space becomes available in the volatile log, message data is copied to the log and the write-protection is removed from the corresponding pages. Blocking the application is necessary only when it tries to write to one of the protected pages. The application has to wait for space for the message to become available again in the log. This scheme extends the message log into the address space of the application and reduces the probability of overflowing the volatile log. For `sparse`, the message logging overhead of an implementation that does not employ this technique increases from 2.5% to 26% with SBML. With the increase in processor speeds, it is reasonable to expect that the applications will produce more messages, and therefore this optimization will gain in importance.

4.3.3 SBML versus RBML

Comparing the columns labeled "SBML Logging" and "RBML Logging" in Table 3, we see that sender-based message logging is more efficient than receiver-based logging, especially for communication intensive programs like `sparse`. There are two reasons for this phenomenon. First, because the message can be logged after it is transmitted to the network, logging a message at the sender is not in the critical path of interprocess communication. Second, logging at the sender reduces the amount of messages to be logged for applications such as `gauss` and `sparse`. In these applications, one processor typically broadcasts the result of a single iteration to the rest. Therefore, in RBML the same data is logged at each receiver, while only a single copy is logged at the sender for SBML. It follows that the amount of logged data is smaller for SBML than RBML. These results quantitatively confirm previous qualitative claims about this issue [18, 35].

5 Message Logging with Coordinated Checkpointing

The results described in Section 4 led us to consider a new message logging design. Given that message logging offers the lowest output latency, and that coordinated checkpointing offers the best failure-free performance in addition to limited recovery time and simplified garbage collection, we propose a design that combines message logging with *coordinated* rather than with independent checkpointing. This design maintains the low output latency of message logging, but, in addition, it allows an important optimization that improves the failure-free performance over the traditional design that uses independent checkpointing. Specifically, we show that for SBML and Manetho the message log need not be written to stable storage, reducing the logging overhead and implementation complexity.

In Sections 5.1 and 5.2 we discuss the new protocol and its advantages. In Section 5.3 we compare the performance to the original design. We focus on SBML and Manetho. Coordinated checkpointing can also be combined with RBML, but the message logs would still need to be written to stable storage, reducing the performance advantage (the volatile log in RBML is not of much use if the receiver fails!). In addition, SBML has shown to be superior to RBML.

5.1 New Protocols

Both new versions of SBML and Manetho take coordinated checkpoints instead of independent checkpoints. The protocols can use *any* algorithm for coordinating the checkpoints [14, 23, 24, 27, 32, 34, 37]. In our implementation, we reuse the coordinated checkpointing protocol that we used before in our study of coordinated checkpointing [11]. The recovery algorithm for each protocol remains unchanged. The corresponding correctness proofs simply carry over to the new design [10, 12, 35], since they assume arbitrary sets of process checkpoints of which a consistent set is a special case. The new versions perform the same dependency tracking as the original designs, and the output commit algorithms are also identical.

5.2 Advantages

By using coordinated checkpointing we avoid having to write the message log to stable storage. In Manetho, in particular, a process writes its volatile (sender) log to stable storage when it takes an independent checkpoint [12]. This is necessary to guarantee that a process never rolls back beyond its latest checkpoint. Indeed, if a message sent before the checkpoint was not logged, and both the sender and the recipient of that message failed, then it would be necessary to roll back the sender to an earlier checkpoint in order to recreate the message. For SBML, the log does not have to be saved on each checkpoint, but nonetheless it must be saved periodically to *stable* storage to guarantee that the maximum recoverable state always advances. It also must be saved to stable storage when the volatile log overflows. Strom *et al* describe a way of compressing the log without coordinating the checkpoint, but their scheme requires a large amount of communication between the processes to determine which messages may be removed [35]. Wang and Fuchs describe several algorithms that can be used to eliminate messages that will never be needed for recovery, but they still require multihost coordination to determine these "obsolete" messages [40, 41].

With coordinated checkpointing, saving the log at the time of the checkpoint is no longer necessary. Since no process will ever roll back to a state before its checkpoint, the only messages that could ever be required for replay are those that are "in transit" at the time of the checkpoint, i.e., messages sent before the sender's checkpoint but received after the receiver's checkpoint. Some coordination protocols consider such a message to be part of the checkpoint of the receiver, and therefore there is no special handling required [8, 14, 23]. Other protocols require such messages to be identified and treated as messages from the outside world that must be logged on stable storage before the receiver may send another message [11, 24]. To handle these situations, each message is tagged with the number of the latest checkpoint taken by the sender. The receiver can then determine that the message is a cross-checkpoint message by comparing the checkpoint number in the message with its local checkpoint number. Our experiments indicate that cross-checkpoint messages occur very infrequently in practice.

An overflow in the volatile log may still require the log to be written to secondary storage, but the important distinction is that it does not need to be written to *stable* storage. Stable storage is typically replicated for high availability and is provided over the network. For secondary storage, a local disk can be used.

In addition to reduced stable log access, the new protocols benefit from simplified garbage collection. Because a consistent checkpoint establishes a recovery line, no event prior to the last checkpoint will be replayed during recovery. All dependency tracking information relating to events that occurred prior the last checkpoint can thus be removed without an explicit garbage collection protocol, unlike in the original version of each protocol. This results in a simpler implementation since no explicit code to handle garbage collection is necessary, and also better performance and lower storage overhead. Finally, the original design of the SBML protocol requires each process to maintain several checkpoints on stable storage [35].‡ In the new version, each process need not maintain more than one permanent checkpoint on stable storage. This checkpoint also establishes a bound on the time for recovery for the SBML protocol.

5.3 Performance

We implemented the new versions of each protocol and compared their performance to the original implementations. Table 5 shows a comparison between the failure-free overhead of the two versions of each protocol. Again, no garbage collection was performed in the old versions of both protocols, and therefore the results underestimate the performance advantage of the new versions.

The results show that for the applications under study, the combination of message logging with coordinated checkpointing has a performance edge over traditional designs. For the applications with a considerable communication load, performance benefits from the reduction in stable storage access.

For `sparse`, the benefits of the new method diminish with increasing checkpointing intervals. For short intervals, no overflow occurs in the volatile message log between checkpoints, so the log is never written to secondary storage. The performance differences reflect the reduced stable storage access. For longer intervals, the volatile logs overflow and they have to be written to a secondary storage device. Our workstations were, unfortunately, diskless, and thus secondary storage had to be provided by a network file server. As a result, the cost of secondary storage access on log overflow was not very different from the cost of stable storage access on a checkpoint, and the performance differences between the two methods are small. If the log were to be written to a local disk, then the performance differences would remain, even for longer checkpointing intervals.

5.4 Discussion

We have presented new versions of two message logging protocols. The new design departs from traditional ones in the combination of message logging and coordinated checkpointing. This new design offers the same output commit latency as conventional message logging, but improves the failure-free execution time by avoiding the maintenance of a message log on stable storage, simplifies garbage collection, limits rollback to the last checkpoint, and improves the utilization of space on the stable storage device. In light of these results, our work can be viewed as an enhancement of message logging protocols. Alternatively, it can be viewed as an addition to existing coordinated

‡The original design of Manetho provides for bounded recovery time because of its dependency tracking [12].

Program Name	% Increase in running time				
	Checkpointing Interval	SBML		Manetho	
		Coord.	Indep.	Coord.	Indep.
gauss	2 min.	0.3	1.0	1.1	1.3
	5 min.	0.2	0.5	1.1	1.2
	10 min.	0.1	0.2	1.0	1.1
grid	2 min.	1.8	1.9	2.6	2.8
	5 min	0.6	0.6	1.3	1.6
	10 min.	0.3	0.4	1.0	1.3
sparse	2 min.	2.0	4.1	3.8	5.6
	5 min	3.0	3.2	4.0	4.4
	10 min.	2.8	2.8	3.8	3.8
tsp	2 min.	0.0	0.0	0.0	0.0
	5 min	0.0	0.0	0.0	0.0
	10 min.	0.0	0.0	0.0	0.0

Table 5 A comparison between the failure-free performance overhead of the two versions of SBML and Manetho.

checkpointing protocols, where message logging is added to provide efficient interactions with the outside world for those applications where tracking nondeterminism can be done efficiently.

6 Related Work

Many message logging protocols have been proposed in the literature [2,3,5,6,13,15–20,22,28–30,33,35,36]. To the best of our knowledge, our paper is the first to advocate the use of coordinated checkpointing as the method of choice for saving processes' states in message logging protocols. We are also not aware of any work that compares message logging protocols with coordinated checkpointing methods on the same software and hardware platforms. This comparative analysis revealed that message logging systems can benefit from the use of coordinated checkpointing. It has also shown that in implementing message logging systems, sender-based logging is the method of choice. Other researchers have advocated sender-based logging [18, 35] on qualitative grounds, but we believe our results are the first to provide quantitative evidence supporting this argument.

Many schemes for coordinated checkpointing have appeared in the literature [1, 7, 14, 21, 23–27, 32, 34, 37–39]. To the best of our knowledge, none of these systems can handle interactions with the outside world except by taking a consistent checkpoint. Any of these systems can benefit from the addition of message logging to provide better performance when interacting with the outside world.

7 Conclusions

We have presented an experimental study showing that traditional designs where message logging is used with independent checkpointing do not have a performance advantage over systems based solely on coordinated checkpointing. Therefore, the real purpose of message logging is no longer to "fix up" inconsistencies between checkpoints, but rather to reduce the output latency that is present with any rollback-recovery method.

Using the above results, we proposed new protocols that combine message logging with coordinated checkpointing, deviating from traditional systems where independent checkpointing is used. The proposed combination of coordinated checkpointing and message logging leads to better failure-free performance, reduced implementation complexity, more efficient use of space on stable storage, and smaller recovery time compared to traditional protocols. Meanwhile, the new schemes maintain the small output commit latency of message logging. Thus, the choice between message logging and coordinated checkpointing should depend on whether the application interacts with the outside world or not, and on the efficiency of tracking nondeterminism.

Our study also highlighted three issues that are of interest to future implementors of message logging protocols. First, we showed that memory management techniques such as copy-on-write are becoming necessary to avoid overflowing the volatile log and the ensuing performance penalty. Second, we showed that for committing output it suffices to write dependency information to stable storage, without writing the message data. Third, we confirmed the superiority of sender-based message logging to receiver-based logging.

Future work should examine the performance of these methods when failures and recoveries are involved. Future work will also examine the above results in different hardware platforms. We conjecture that the results will continue to hold because of current trends in stable storage and networking technologies.

Acknowledgments

We would like to thank the referees for their valuable comments.

References

[1] M. Ahamad and L. Lin. Using checkpoints to localize the effects of faults in distributed systems. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 1–11, October 1989.

[2] D.F. Bacon. File system measurements and their applications to the design of efficient operation logging algorithm. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 21–30, October 1991.

[3] J. F. Bartlett. A Non Stop kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 22–29, December 1981.

[4] B. Bhargava, S-R. Lian, and P-J. Leu. Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms. In *Proceedings of the International Conference on Data Engineering*, pages 182–189, March 1990.

[5] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 90–99, October 1983.

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

[7] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of the 4th Symposium on Reliable Distributed Systems*, pages 207–215, October 1984.

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

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

[10] E.N. Elnozahy. *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication.* PhD thesis, Rice University, October 1993. Also available as Technical Report TR93-212.

[11] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[12] E.N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers Special Issue On Fault-Tolerant Computing*, 41(5):526–531, May 1992.

[13] A. Goldberg, A. Gopal, K. Li, R. Strom, and D. Bacon. Transparent recovery of Mach applications. In *Proceedings of the Usenix Mach Workshop*, pages 169–184, October 1990.

[14] S. Israel and D. Morris. A non-intrusive checkpointing protocol. In *The Phoenix Conference on Communications and Computers*, pages 413–421, 1989.

[15] P. Jalote. Fault tolerant processes. *Distributed Computing*, 3:187–195, 1989.

[16] D.B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* PhD thesis, Rice University, December 1989.

[17] D.B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, October 1993.

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

[19] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181, August 1988.

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

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

[22] K.H. Kim, J.H. You, and A. Aboulnaga. A scheme for coordinated execution of independently designed recoverable distributed processes. In *Proceedings of the 16th International Symposium on Fault-Tolerant Computing*, pages 130–135, 1986.

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

[24] T.H. Lai and T.H. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, May 1987.

[25] L.A. Laranjeira, M. Malek, and R. Jenevien. Space/time overhead analysis and experiments with techniques for fault tolerance. In *Third IFIP International Working Conference on Dependable Computing for Critical Applications*, pages 175–184, September 1992.

[26] P. Leu and B. Bhargava. Concurrent robust checkpointing and recovery in distributed systems. In *Proceedings of the International Conference on Data Engineering*, February 1988.

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

[28] A. Lowry, J.R. Russell, and A.P. Goldberg. Optimistic failure recovery for very large networks. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 66–75, October 1991.

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

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

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

[32] L.M. Silva and J.G. Silva. Global checkpointing for distributed programs. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 155–162, October 1992.

[33] A.P. Sistla and J.L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 223–238, August 1989.

[34] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 382–388, May 1986.

[35] R.E. Strom, D.F. Bacon, and S.A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, pages 44–49, June 1988.

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

[37] Y. Tamir and C.H. Séquin. Error recovery in multicomputers using global checkpoints. In *1984 International Conference on Parallel Processing*, pages 32–41, August 1984.

[38] Z. Tong, R.Y. Kain, and W.T. Tsai. A lower overhead checkpointing and rollback recovery scheme for distributed systems. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 12–20, October 1989.

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

[40] Y.-M. Wang and W.K. Fuchs. Optimistic message logging for independent checkpointing in message-passing systems. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 147–154, October 1992.

[41] Y.-M. Wang and W.K. Fuchs. Scheduling message processing for reducing rollback propagation. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 204–211, July 1992.