On the Relevance of Communication Costs of Rollback-Recovery Protocols

E.N. Elnozahy

June 1995

CMU-CS-95-167

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in the Proceedings of the SIGACT-SIGOPS Symposium on Principles of Distributed Computing, August 1995.

Author was supported in part by the National Science Foundation under grants CCR 9410116 and CCR 9502933, and by the Advanced Research Project Agency under contract DABT63-93-C-0054

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the NSF, ARPA or the U.S. Government.

Keywords: Algorithms, Checkpointing, Distributed Systems, Fault tolerance, Logging, Rollback-Recovery

Abstract

Communication overhead has been traditionally the primary metric for evaluating rollback-recovery protocols. This paper reexamines the prominence of this metric in light of the recent increases in processor and network speeds. We introduce a new recovery algorithm for a family of rollback-recovery protocols based on logging. The new algorithm incurs a higher communication overhead during recovery than previous algorithms, but it requires less access to stable storage and imposes no restrictions on the execution of live processes. Experimental results show that the new algorithm performs better than one that is optimized for low communication overhead. These results suggest that in modern environments, latency in accessing stable storage and intrusion of a particular algorithm on the execution of live processes are more important than the number of messages exchanged during recovery.

1 Introduction

The cost of communications has been traditionally the primary metric for evaluating distributed rollback-recovery protocols [16, 17]. This cost is typically expressed as a function of the number of nodes in the system, describing the number of required messages and their sizes. The prominence of this metric is due in part to the high cost of interprocess communication in computer networks at the time when many of these protocols were incepted. It also provides a precise yardstick against which different protocols can be compared. Other qualitative factors were also considered, but they were deemed less important given the practical constraints imposed by old technology. Examples of these secondary factors include the cost of accessing stable storage during recovery, whether a protocol requires live processes to block or refrain from exchanging messages during recovery, and whether a failure may result in some live processes becoming orphans [17].

In this paper, we reexamine the prominence of communication costs as the primary metric for evaluating rollback-recovery protocols. The advances in network and processor technologies and the relative increase in the penalty of accessing stable storage are at odds with many premises that were valid when existing protocols were published. Therefore, we argue that other metrics become more important such as the effect of accessing stable storage and the protocol's intrusion on the operation of live processes during recovery. To support this argument, the paper presents a new recovery algorithm that incurs additional control messages to allow live processes to continue operation with no interference or blocking to access stable storage during recovery. The algorithm can be used in several rollback-recovery protocols based on logging. It does not depend on the particular technique used to gather dependency information during failure-free operation. To demonstrate this, we present the algorithm in the context of the Family-Based Logging protocols [2, 3, 4]. This family of protocols is a parameterized presentation of a set of recovery protocols that differ in the degree of failures they tolerate. We will describe the salient features of this protocol family as they relate to the purpose of this paper.

We compare the performance of a prototype implementation of this algorithm to another algorithm that is optimized to reduce the communication overhead. The comparison shows that the new algorithm performs as well as or better than the message-optimal algorithm, depending on the number of failures. These results suggest that in light of recent advances in technology, supposedly secondary factors in evaluating rollback-recovery protocols are at least as important as the communication overhead. These factors include the effect of recovery on live processes beyond whether the protocol does or does not protect them from becoming orphans. It is hoped that theoretical formulations could be developed to precisely express the effects of these factors in the same way that message complexity became the yardstick for evaluating and comparing these protocols.

The rest of this paper is organized as follows. We describe the family based logging protocols on which we base this research in Section 2. An informal presentation of the recovery protocol and proofs of correctness appear in Sections 3 and 4. The results of the experiments follow in Section 5. We review related work in Section 6 and conclude in Section 7.

2 Family-Based Logging Protocols

We assume a distributed, asynchronous system where deterministic processes may fail by crashing. The new recovery algorithm is explained in the context of the Family Based Logging protocols (FBL) [2, 3, 4]. These protocols are based on a simple idea: To tolerate f process failures in a rollback-recovery system, it is sufficient to log each message in the volatile store of its sender and to log its receipt order in the volatile store of $f+1$ different hosts. This simple idea is the basis of a powerful family of protocols that possess several desirable properties. First, no logging to stable storage is necessary except in the case $f = n$, where n is the number of hosts in the system. Therefore, applications pay only the overhead that corresponds to the number of failures they are willing to tolerate. Second, FBL protocols protect the live processes from the effects of failures. Specifically, no live process becomes an orphan because of a failure in another process [17]. Third, FBL protocols do not require any blocking during failure-free operation, thereby reducing the performance overhead.

In summary, FBL protocols have the low-overhead properties of optimistic protocols without having to make any optimistic assumptions. They can be thought of as a parameterized generalization over previous rollback-recovery protocols. For example, the instance of FBL where $f = 1$ corresponds to a variation on Sender-Based Message Logging [11], while the instance where $f = n$ corresponds to the Manetho protocol [8]. For this reason, we picked FBL to ensure the generality of the results presented here.

2.1 An Example

To illustrate the operation of FBL protocols, consider the example shown in Figure 1. In this example, there are three processes p, q and r represented by the vertical lines. Process p receives a message m and then sends a message m' to process q, which in turns sends a message m'' to process r. For simplicity, the figure does not show the sender of message m. Consider the operation of the FBL protocols for $f = 2$ in the context of this example. According to the definition of FBL protocols, message m' is a descendent of message m, and message m'' is a descendent of message m'.

During failure-free operation, each process piggybacks on each application message the receipt orders of its direct and transitive descendents. The receiver of the message will record these receipt orders in its volatile log. The message

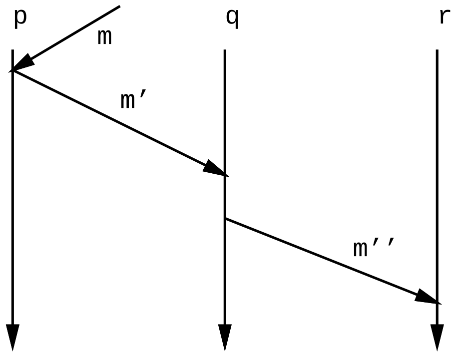

Figure 1: An example execution.

data is logged in the volatile log of the sender, and is used for replay during recovery if needed. Propagation of the receipt order of a certain message stops as soon as it has been recorded in $f+1$ hosts. Therefore, the receipt order of m need not be propagated further than r for $f = 2$ in the example shown in Figure 1. The actual protocol is more complicated than this simple description implies, and the reader is referred to the appropriate reference for a full explanation [4].

Now, consider what happens when process p fails. To recover to a state consistent with the rest of the system, process p needs to receive message m in the same order as before the failure. The receipt order of the message is available in the volatile log of either process q or r. The data of the message is available in the log of its sender. Therefore, process p has the necessary information to recover to a state consistent with the rest of the system. The same argument extends up to two failures. Assume for instance that processes p and q fail. To recover to a state consistent with r, the recovering processes receive the required receipt orders of messages m and m' from r. Process p receives m from its sender, and because the execution is deterministic, it will reproduce m' for the benefit of the recovery of process q.

2.2 Protocol Intrusion and its Effects

The recovery algorithm presented in the original description of FBL protocols requires live processes to block while recovery is in progress [4]. This intrusion has the undesirable property of increasing the relative cost of a process failure and its impact on application performance. Many published protocols for rollback-recovery based on logging share this undesirable property [8, 9, 10, 12, 13, 16, 17].

To see why this intrusion is necessary, consider the example in Figure 1. Assume that process p fails after sending message m'. To recovery from the failure, process p restarts from a previous checkpoint and enlists the help of other processes to determine the receipt order of each message that it has received before the failure and replay it. If process r receives the request from p before receiving m'', then it will not tell p about the receipt order of message m. Now assume that process q fails after sending message m'' but before responding to p's request. Thus, process q loses the information about m's receipt order and after it restarts, it will not be able to convey any information about m to p. Process p thus has received replies from all live processes, none of which indicates the receipt order of message m. When process r receives message m'' it will be inconsistent with the state of process p as m'' reflects the receipt of message m by p. The fundamental problem here is that failures that occur during recovery may throw the system into an inconsistent state.

Previous protocols resorted to different ways to solve this problem. For example, one protocol simply blocks the execution of the entire system pending the outcome of the recovery phase [12]. Another protocol prevents live processes from accepting any application message that could potentially create an inconsistency with its reply to a recovering process until the latter completes recovery [8]. Furthermore, this protocol requires each live process to record its replies to recovery requests on stable storage before it can send them, introducing further delays. In the example above, this protocol requires that process r recognize that message m'' is inconsistent with its reply to process p and therefore it refrains from consuming this message until p announces the set of messages that it was able to recover. Process r would then discard the message if p was unable to recover message m, or it would deliver it to the application otherwise. There are two problems with this approach. First, it may introduce unnecessary delays in delivering legitimate messages that would not create any inconsistencies. In the example above, if process p was able to get the information about message m from process q, then process r would be safe to consume the message as soon as it arrives. Second, the implementation of such a protocol requires non-trivial modifications to the communication protocol to recognize the potentially unsafe messages [7]. A complex implementation reduces the performance of the communication subsystem during failure-free operation and reduces the confidence in its robustness.

To appreciate the effect of intrusion on live processes, consider again the example shown in Figure 1. Assume that as soon as process r receives the recovery request from process p, the protocol blocks the progress of process r or prevents it from receiving application messages that could potentially lead to inconsistencies. Consider the case when process q fails while process p is recovering. A typical implementation would require several seconds of timeouts and retrials to detect that process q has indeed failed. A recovery procedure would then reload the saved state of process q from stable storage. In light of the relatively high cost of stable storage access, and depending on the size of process q, restoring its state may take tens of seconds or a few minutes. Meanwhile, process r is unable to progress or is prevented from consuming application messages that would allow it to progress. Thus, if consuming the message would not result in any inconsistencies, or if the protocol altogether requires blocking of the live processes, then process r would unnecessarily waste tens of seconds or even minutes because of failures in other processes. Clearly, the situation would worsen in a larger system where a few simultaneous failures may occur. Thus, the effect of a failure on the application progress is not confined to the processes that fail, it extends to the live processes as a result of the intrusion required to guarantee safety. This fact is unfortunate given that protocols such as FBL go to a great length in shielding the live processes from the effects of failures.

The algorithm in the following section solves this problem and can be applied to any instance of the FBL protocols. It does not depend on the particular method used to propagate and maintain the receipt orders of the messages during failure-free operation. Thus the results are applicable to other protocols which are instances of FBL [8, 11, 13].

3 The New Algorithm

3.1 Rationale and Motivations

A recovering process needs a consistent snapshot of the message receipt order information that is scattered throughout the system. This snapshot should be consistent despite failures that may occur during recovery. This fact does not fundamentally necessitate any blocking or intrusion on the execution of live processes. Furthermore, the cost of communication is not at all expensive in modern systems. This fact is consistent with the current technological trends and will likely continue for a while. We therefore contend that communication overhead should not be the sole target for optimization. Indeed, we strive to reduce the need for stable storage access and interference with live processes during recovery, even at the expense of additional messages.

3.2 Data Structures

Each process maintains the following variables:

state : This variable indicates whether the process is *live*, *recovering*, or is a *recovery leader*.

incarnation : This is an integer that is incremented by one each time a process recovers from a failure. Each process tags every application message it sends with its **incarnation** value. A receiver rejects any message that originates from a previous incarnation of its sender.

incvector : This vector contains the incarnation numbers of the other processes in the system.

depinfo : This is an abstract presentation of the message receipt order information that is maintained by the process. It could take the form of dependency vectors [17], a dependency matrix [4, 12], or a dependency graph [8].

R: The set of failed processes that are recovering simultaneously with the process.

L: The set of live processes.

ord: A system-wide monotonic number that is incremented whenever a process starts recovery. The process whose recovery corresponds to the lowest value becomes the *recovery leader*.

3.3 Informal Description

For the case where $f = n$ we model stable storage as an additional process that never fails or sends a message. If a process p fails, it will restore a previous checkpoint and increment its incarnation number. The process also acquires an ordinal number for its recovery, and it becomes a recovery leader if no "recovering" process has a lower number. Otherwise, it sets its state to "recovering" and blocks until the current recovery leader completes the algorithm. If the process is a recovery leader, it will broadcast a request for the incarnation number to every "recovering" process. Each "recovering" process replies with its incarnation number and the recovery leader updates its **incvector** accordingly. After receiving replies from all recovering processes, the recovery leader sends a request to all live processes requesting their **depinfo** information. The request includes **incvector**. Each live process sends a reply containing its **depinfo** and updates its **incvector**. Using **incvector**, a live process stops receiving stale messages that originated before the failure of any recovering process. This step ensures that after replying to the recovery leader, a live process will not acquire a dependency on a stale message that may throw the system into an inconsistent state. After gathering all the **depinfo** data, the recovery leader sends it to each recovering process. The recovering processes use this data to replay the execution and recover to a consistent state [4].

If a live process fails before replying to the recovery leader, the latter restarts the gathering of the **depinfo** data by resending the **depinfo** request after updating **incvector**. This step ensures that the recovery leader gathers a consistent snapshot of the **depinfo** data despite concurrent failures during recovery. If the recovery leader fails, then the next process in ordinal number becomes a recovery leader and restarts the algorithm.

The algorithm does not restrict the progress of live processes by blocking them or restricting their message interactions. Live processes continue to process the messages they receive if the incarnation number tagging the message is valid. This step is necessary in any environment where messages can be delayed.

3.4 Description

The algorithm for the recovery leader follows. Note that steps 1 through 3 are executed by each recovering process including the leader.

1. Restore state;

2. $incarnation \leftarrow incarnation + 1$;

3. $ord \leftarrow ord + 1$;

4. **for** each process $q \in R$ **do**
   $incvector[\mathrm{q}] \leftarrow q.incarnation$;

5. **for** each process $q \in L$ **do**
   if q failed **then goto** 4;
   $depinfo \leftarrow q.depinfo$;
   $q.incvector \leftarrow incvector$;

6. **for** each process $q \in R$ **do**
   $q.depinfo \leftarrow depinfo$;

4 Correctness Proof

4.1 Definitions

A message m is an antecedent of message m' if m *happens before* m' [14]. Message m' is also called a descendent of m. Figure 1 shows this relationship. Call a message m *visible* if there exists a live process that received it.

4.2 Termination

The algorithm restarts whenever a live process fails before responding to the recovery leader's **depinfo** request. Since no more than f failures occur, the algorithm cannot restart more than f times. The algorithm also does not block any live process, and recovering processes block only until the recovery leader finishes gathering the **depinfo** data.

4.3 Safety

To establish the safety of the algorithm, the resulting system state after recovery must show all antecedents of visible messages as received. We show this property by contradiction. Assume that the algorithm cannot determine the receipt order of a message m which is an antecedent of a visible message m'. Let process p be the receiver of m'. There are two cases to consider:

1. The receipt order information piggybacked on m' contains the receipt order of m. There are two cases to consider:

   - Process p receives the message before responding to the recovery leader's request. Therefore, process p sends this information to the recovery leader which makes it available to the receiver of m. This remains true if the recovery leader fails and another process restarts the algorithm.
   - Process p receives the message after responding to the recovery leader's request. The message's sender cannot be a process that failed because **incvector** announced by the leader would force p to reject the message. Also, the message's sender cannot be a process that fails during recovery because the leader would restart the algorithm and p will send the receipt order to the leader. Therefore, the message sender must be a live process. A simple induction on the number of live processes shows that at least one live process will send the receipt order to the leader.

2. The receipt order information piggybacked on m' does not contain the receipt order of m. In this case, the receipt order of the message must be stored at $f+1$ processes, and at least one of them never fails and will always return the required information to the recovery leader.

Thus, it is not possible to have a visible message without being able to recover the receipt orders of all its antecedents, contradicting the hypothesis.

4.4 Liveness

To establish the liveness of the algorithm, the failed processes must recover to a state where they have sent all visible messages. The proof proceeds by induction on n, the number of antecedents of a visible message.

1. Base case, $n = 0$: In this case, the message does not have an antecedent and because the execution of the sender is deterministic, it will regenerate the message.

2. Induction case, assume true for n: For $n+1$, all antecedents of the message have been sent except for the immediate antecedent. But the visible message has the receipt order of its immediate antecedent or it is available because it has been logged at $f+1$ processes. Therefore, the immediate antecedent can be replayed to its receiver (the sender of the visible message) in the specified order and because of the deterministic execution, the visible message will be regenerated.

5 Evaluation

We implemented a prototype of the new recovery algorithm on a network of eight DEC 5000/200 workstations connected by a 155 Mb/sec ATM network. Each machine is equipped with a 25 MHz MIPS 3000 processors and 32 Mbytes of memory. The size of a process was about one Mbytes. For the purpose of comparison, we also implemented a prototype of a blocking recovery algorithm. In this algorithm, live processes block while recovery takes place.

For a single failure, the recovering process took the same time to recover under both algorithms. However, the blocking algorithm caused each live process to block for about 50 milliseconds on average, while the new algorithm did not affect the execution of the live processes.

In a second experiment, a process failed during the execution of the recovery of another process that failed earlier. Under the two algorithms, the two recovering processes required essentially about five seconds to recover. Most of this time was spent in failure detection and in restoring the state of the second process. The blocking algorithm required each live process to block for the same amount of time, while the new algorithm did not require such blocking. The extra communication overhead required by the second phase of the new algorithm was negligible (about milliseconds) compared to the time that required for failure detection and to restore the state of the second process.

6 Related Work

Rollback-recovery protocols based on logging have been the subject of active research [12, 13, 16, 17]. Most of this work has focused on the behavior of the system during failure-free operation, motivated by the understandable desire to optimize the performance for the most common case. Little is understood about the behavior of these protocols during failures. Unlike previous work, this paper focuses on understanding the performance and behavior of these protocols during recovery.

The new recovery algorithm introduced in this paper does not require live processes to block or refrain from receiving messages during recovery. Previous protocols typically requires either that the entire system block or that live processes refrain from receiving certain application messages. Furthermore, a live process need not perform a synchronous write operation on stable storage during recovery as many recovery protocols require.

Rollback-recovery protocols based on logging have been long classified as either optimistic or pessimistic [17]. Optimistic protocols reduce the overhead of tracking dependencies during failure-free operation at the expense of complicating recovery and the potential for processes that survive failures to become orphans [10, 12, 13, 16, 17]. Pessimistic protocols, on the other hand, simplify recovery and insulate live processes from the effects of failures, at the expense of higher overhead for dependency tracking during failure-free operation [5, 15]. A new generation of protocols emerged that combine the advantages of both classes without the disadvantage. The Manetho protocol protects the live processes from the effect of failures without incurring large overhead to track dependencies [8]. The protocol is designed to tolerate an arbitrary number of failures. FBL protocols introduced a family of parameterized logging protocols that can tolerate a variable number of failures [2, 3, 4]. Thus, the application pays only the price of providing the desired degree of fault tolerance. These protocols combine the advantages of pessimistic and optimistic protocols. The generalized description of these protocols is desirable in that it allows the expression of other protocols by simply picking the correct parameter. For example, the Manetho protocol is an instance of FBL protocols with $f = n$, where n is the total number of nodes in the system.

Previous optimistic and FBL protocols affect the progress of live processes during recovery. Most optimistic protocols block the execution of the live processes while recovery is ongoing. This is perhaps acceptable or even desirable in optimistic protocols since some of these live processes may become orphans. Continuing the execution of a process that may become an orphan makes little sense since it will be aborted anyway. Furthermore, allowing a live process that may become an orphan to communicate with other processes will likely increase the number of orphans.

FBL protocols (as published) also affect the progress of the live processes during recovery. The Manetho protocol requires that live processes refrain from accepting certain application messages while recovery is ongoing, and requires some synchronous logging to stable storage during recovery. The protocol published by Alvisi and Marzullo requires blocking the live processes to ensure safety [1]. This is an unfortunate fact, because FBL protocols go to a great length to protect live processes from becoming orphans. The new recovery algorithm presented in this paper solves this problem. It allows the application at live processes to progress regardless of the failures that occur, including those that occur during recovery. The price to be paid for this advantage is the higher overhead in communication. Evaluation shows that this overhead is not substantial, and we believe that this will continue to be the case given the current trends of increasing processor speeds and network bandwidth.

7 Conclusions

The paper reexamined the tradition of using the communication overhead as the primary metric for evaluating rollback-recovery protocols. Recent increases in network and processor speeds are promoting other factors such as stable storage access and the intrusion of the protocol on the execution of live processes. To support this argument, the paper presented a new algorithm that imposes no restrictions on the execution of live processes during recovery at the expense of a higher communication overhead compared to previous protocols. Live processes do not need to block or refrain from receiving messages during recovery. The algorithm is applicable to a family of log-based rollback-recovery protocols that differ in the number of failures they tolerate. An experimental study showed that the new algorithm performs as well as or better than another protocol that is designed to minimize the communication overhead. These results suggest a rethinking of our evaluation methods to include the effects of technological trends. It is hoped that theoretical formulations could be developed to precisely express the effects of these factors in the same way that message complexity became the yardstick for evaluating and comparing these protocols.

Acknowledgments

I would like to thank the anonymous referees for their constructive comments and suggestions. I would like also to thank Lorenzo Alvisi and Keith Marzullo for answering all my questions.

References

[1] L. Alvisi. Private communication, February 1995.

[2] L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and orphan-free message logging protocols. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, June 1993.

[3] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal and optimal. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, May 1995.

[4] L. Alvisi and K. Marzullo. Trade-offs in implementing optimal message logging protocols. In *Submitted to the International Symposium on Fault Tolerant Computing Systems*, 1995.

[5] 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.

[6] 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.

[7] 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 TR-93-212.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[15] 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.

[16] 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.

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