The Cost of Recovery in Message Logging Protocols

Sriram Rao, **Lorenzo Alvisi**, and **Harrick M. Vin**
Department of Computer Sciences
The University of Texas at Austin
Taylor Hall 2.124, Austin, Texas 78712-1188, USA
E-mail: {sriram,lorenzo,vin}@cs.utexas.edu, Telephone: (512) 471-9792, Fax: (512) 471-8885
URL: http://www.cs.utexas.edu/users/{sriram,lorenzo,vin}

Abstract

Message logging is a popular technique for building low-overhead protocols that tolerate process crash failures. Past research in message logging has focused on studying the relative overhead imposed by pessimistic, optimistic, and causal protocols during failure-free executions. In this paper, we give the first experimental evaluation of the performance of these protocols during recovery. We discover that, if a single failure is to be tolerated, pessimistic and causal protocols perform best, because they avoid rollbacks of correct processes. For multiple failures, however, the dominant factor in determining performance becomes *where* the recovery information is logged (i.e. at the sender, at the receiver, or replicated at a subset of the processes in the system) rather than *when* this information is logged (i.e. if logging is synchronous or asynchronous). From our results, we distil a few lessons that can guide the design of message-logging protocols that combine low-overhead during failure-free executions with fast recovery.

Technical Report TR-98-02, Department of Computer Sciences, University of Texas, Austin, TX.

1 Introduction

Message-logging protocols (for example, [1, 4, 5, 9, 12, 13, 14, 18, 20, 21, 22, 23]) are popular techniques for building systems that can tolerate process crash failures. These protocols are built on the assumption that the state of a process is determined by its initial state and by the sequence of messages it delivers. In principle, a crashed process can be recovered by (1) restoring the process to its initial state and (2) rolling it forward by re-playing to it messages in the same order they were delivered before the crash. In practice, message logging protocols limit the extent of roll-forward by having each process periodically save its local state in a checkpoint. The delivery order of each message is recorded in a tuple, called the message's *determinant*, which the delivering process logs on stable storage. If the determinants of all the messages delivered by a crashed process are available during recovery, then the process can be restored to a state *consistent* with the state of all operational processes. Two states s_p and s_q of processes p and q are consistent if all messages from q that p has delivered during its execution up to state s_p were sent by q during its execution up to state s_q, and vice versa. An *orphan* process is an operational process whose state is inconsistent with the recovered state of a crashed process. All message-logging protocols guarantee that upon recovery no process is an orphan, but differ in the way they enforce this consistency condition:

- Pessimistic protocols [4, 12, 18] require that a process, before sending a message, synchronously log on stable storage the determinants and the content of all messages delivered so far. Thus, pessimistic protocols never create orphan processes. However, synchronously logging determinants on stable storage imposes a significant overhead during failure-free executions.

- Optimistic protocols [5, 13, 14, 20, 21, 23] allow processes to communicate even if the determinants they depend upon are not yet logged on stable storage. These protocols only require that determinants reach stable storage eventually. Optimistic protocols can perform very efficiently in failure-free executions. However, if any of the determinants are lost when a process crashes, then orphans may be created. To reach a consistent global state, these processes must be identified and rolled back.

- Causal protocols [1, 9] combine some of the positive aspects of pessimistic and optimistic protocols: They never create orphans, yet they do not write determinants to stable storage synchronously. In causal protocols, determinants are logged in volatile memory. To prevent orphans, processes piggyback their volatile log of determinants on every message they send [1]. This guarantees that if the state of an operational process p causally depends [15] on the delivery of a message m, then p has a copy of m's determinant in its volatile memory. This property is sufficient to restore a crashed process in a state consistent with the state of all operational processes.

Although several studies have measured the overhead imposed by each of these approaches during failure-free executions [7, 10], their merits during recovery have been so far argued mostly qualitatively. For instance, there is consensus that pessimistic protocols are well-suited for supporting fast recovery, since they guarantee that all determinants can be readily retrieved from stable storage. The opinions about optimistic protocols are less unanimous. On the one hand, these protocols seem unlikely candidates for fast recovery because, to restore the system to a consistent state, they require to identify, roll

[1]If there exists an upper bound f on the number of concurrent crashes and processes fail independently, then a determinant logged by $f + 1$ processes does not need to be piggybacked further.

back, and then roll forward all orphan processes. On the other hand, recent optimistic protocols employ techniques for quickly identifying orphans and can roll forward orphans concurrently, thereby reducing recovery time.

Although the literature contains careful analyses of the cost of recovery for different optimistic protocols in terms of the number of messages and the rounds of communication needed to identify and roll back orphan processes [5, 7, 11, 13, 20, 21, 23], in general no experimental evaluations of their performance during recovery are offered.

The performance of causal protocols during recovery has also been debated. Proponents of these protocols have pointed to the fact that causal protocols, like pessimistic protocols, never create orphans and therefore never roll back correct processes. However, with causal protocols a process can start its recovery only after collecting the necessary determinants from the volatile logs of the operational processes. It has been qualitatively argued [5] that optimistic protocols that start recovery without waiting for data from other processes may have a shorter recovery time than causal protocols.

Finally, little is known about the effect of changes in f, the number of concurrent process failures, on the recovery costs of pessimistic, optimistic, and causal protocols.

In the past, the absence of a careful experimental study of the performance of these protocols during recovery could be justified by arguing that, after all, it was not needed. Distributed applications requiring both fault-tolerance and high availability were few and highly sophisticated, and its users could typically afford to invest the resources necessary to mask failures through explicit replication in space [19] instead of recovering from failures through replication in time. As distributed computing becomes commonplace and many more applications are faced with the current costs of high availability, there is a fresh need for recovery-based techniques that combine high performance during failure-free executions with fast recovery.

In this paper, we take an initial step towards the development of these new protocols by presenting the first experimental study of the recovery performance of pessimistic, optimistic, and causal protocols. Contrary to our initial intuition, our results indicate that pessimistic and causal protocols outperform optimistic protocols only when $f = 1$. For $f > 1$, the dominant factor in determining recovery time becomes *where* the recovery information is logged (i.e. at the sender, at the receiver, or replicated at a subset of the processes in the system) rather than *when* this information is logged (i.e. if logging is synchronous or asynchronous). Hence, optimistic protocols, even if suffering from rollbacks, can often outperform implementations of pessimistic and causal protocols that are less efficient in supporting fast retrieval of messages and determinants used during recovery. From our results, we distill a few lessons that can guide the design of future message-logging protocols.

The rest of the paper is organized as follows. In Section 2, we describe our implementation of message logging protocols and checkpointing. We briefly describe the application programs used in this study in Section 3. The experimental analysis of the recovery costs for the pessimistic, optimistic, and causal logging protocols is presented in Section 4. Section 5 discusses a few principles that can be used to design message-logging protocols for fast crash recovery. Finally, Section 6 offers some concluding remarks.

2 Implementation

To measure the cost of recovery in message logging protocols, we have implemented a fault-tolerance layer consisting of a communication substrate, a checkpoint manager, and a message-logging protocol suite. For tolerating hardware failures, processes are named using a name server which provides location-independent names.

- **Communication substrate**: The communication substrate provides interfaces to create and destroy point-to-point FIFO communication channels among cooperating processes, as well as to send and deliver messages. Communication channels are implemented as `tcp` connections.

- **Checkpoint manager**: The checkpoint manager periodically saves on stable storage the state of each process, which includes heap, stack, and data segments, plus the mapping of implicit variables such as program counters and machine registers to their specific values. Stable storage for checkpoints is provided by a highly available network file server. In case of a failure, the failed process is re-started and its state is restored to that recorded in the latest checkpoint.

  In our current implementation, checkpoints are synchronous (i.e., applications block during checkpointing). The checkpointing mechanisms used are similar to those described in [17]. We are currently enhancing our implementation to utilize optimizations such as incremental checkpointing and copy-on-write [8]. Although such optimizations reduce the cost of checkpointing and hence the failure-free execution time of distributed application, they do not affect the failure recovery overhead, which is the focus of this paper.

- **Message-logging Protocol Suite**: This suite contains representative protocols for each of the three styles of message logging:

  - *Pessimistic logging*: We have implemented two pessimistic protocols. The first protocol is *receiver-based*: a process, before sending a message, logs to stable storage both the determinants and the contents of the messages delivered so far. The second protocol is instead *sender-based* [12]: the receiver logs synchronously to stable storage only the determinant of every message it delivers, while the contents of the message are stored in a volatile log kept by the message's sender [2] This protocol is similar to the one described in [22].

    In both of these protocols, the first step of recovering a process p consists in restoring it to its latest checkpoint. Then, in the receiver-based protocol, the messages logged on stable storage are replayed to p in the appropriate order. In the sender-based protocol, instead, p broadcasts a message asking all senders to retransmit the messages that were originally sent to p. These messages are matched by p with the corresponding determinants logged on stable storage and then replayed in the appropriate order.

  - *Optimistic logging*: Among the numerous optimistic protocols that have been proposed in the the literature (for instance [10, 12, 13, 14, 20, 21, 23]), we have implemented the protocol described in [5]. This protocol, in addition to tolerating an arbitrary number of failures and preventing the uncontrolled cascading of rollbacks known as the *domino effect* [21], implements a singularly efficient method for detecting orphans processes. In this protocol, causal dependencies are tracked using vector clocks [16]. On a message send, the sender piggybacks its vector clock on the message; on a message deliver, the receiver updates its vector clock by computing a component-wise maximum with the piggybacked vector clock. The determinants and the content of the messages delivered are kept in volatile memory

[2]Some sender-based pessimistic protocols keep both determinants and message contents at the senders [12, 14]. We have not implemented these protocols because they can only tolerate at most two concurrent failures.

logs at the receiver and periodically flushed to stable storage. Since in a crash these logs in volatile memory are lost, orphans may be created. To detect orphans, a recovering process simply sends a failure announcement message containing the vector clock of the latest state to which the process can recover. On receiving this message, each operational process compares its vector clock with the one contained in the message to determine whether or not it has become an orphan. An orphan process first synchronously flushes its logs to stable storage. Then, it rolls back to a checkpoint consistent with the recovered state of the failed process and uses its logs to roll-forward to the latest possible consistent state.

In our implementation, we have modified the pseudo-code presented in [5] so that the recovering process sends the failure announcements before replaying any message from the log, rather than after all messages in the log have been replayed. This optimization allows the roll-forward of recovering processes to proceed in parallel with the identification, roll-back and eventual roll-forward of orphan processes. As we will see in Section 4, this optimization dramatically improves the performance of the protocol during recovery.

– *Causal logging*: We have implemented the Π_{det} family-based message-logging protocol [2]. This protocol is based on the following observation: in a system where processes fail independently and no more than f processes fail concurrently, one can ensure the availability of determinants during recovery by replicating them in the volatile memory of $f+1$ processes. In our implementation, this is accomplished by piggybacking determinants on existing application messages until they are logged by at least $f+1$ processes [3, 9]. Recovery of a failed process proceeds in two phases. In the first phase, the process obtains its determinants from the volatile logs of the operational processes. In addition, the process also obtains the content of messages it delivered before crashing. This is because in causal protocols, message contents are logged only in the volatile memory of the sender. Also, in the first phase, a recovering process obtains lost messages from the remaining processes. In the second phase, the collected data is replayed, restoring the process to its pre-crash state. To handle multiple concurrent failures, we implemented a protocol that recovers crashed processes without blocking operational processes [6]. In this protocol, the recovering processes elect a leader, which is responsible for collecting determinants and messages on behalf of all recovering processes. The leader then forwards the pertinent data to each recovering process.

3 Applications

For our experiments, we have chosen the following five long-running, compute-intensive applications.

- grid performs successive over-relaxation (SOR) for a Laplace partial differential equation on a grid of 200×200 points. In each iteration, the value of each point is computed as a function of its value in the previous iteration and of the values of its neighbors. The rows of the grid are partitioned using a 1-D decomposition such that the load on all processes is balanced. At the end of each iteration, each process exchanges with its 2 neighbors the new values on the edges of its grid.

- nbody performs an n-body simulation for 625 particles. In the simulation, the motion of a particle depends on the interactive forces between that particle and the remaining particles. Particles are evenly distributed amongst all the processes. During each iteration, each process exchanges the positions of its particles with the other processes in the system.

- gauss performs Gaussian elimination with partial pivoting on a 1024×1024 matrix that represents a system of linear equations of the form $Ax = B$. Each process is initially assigned a subset of the rows of matrix A such that the load on each process is balanced. In each iteration, a process receives a row of the matrix from its predecessor, performs some local computation and sends the row of the matrix it computed to its successor.

- life is the game of life played on a 500×500 grid of points. In each iteration, the value of a grid point is computed as the sum of the values of its 8 neighbors. The rows of the grid are partitioned such that the load on all processes is balanced. At the end of each iteration, each process exchanges with its 2 neighbors the new values on the edges of its grid.

- p2fox performs a predator-prey simulation over a population of rabbits and foxes on a 250×250 grid of points. For the simulation, the grid is evenly divided amongst processes. At the end of each iteration, a process updates the population according to some rules and then exchanges with its 4 neighbors the new values on the edges of its grid.

These applications exhibit different communication patterns. In the grid and life applications, a process communicates mostly with its two neighbors, and in p2fox a process communicates mostly with four of its neighbors. The size of messages exchanged are approximately 2KB. Periodically, however, each process sends 100Byte messages to all the processes in the system. In nbody, each process communicates with all other processes, and the size of these messages is approximately 1KB. In gauss, each process communicates with two of its neighbors, and the size of each message is approximately 15KB.

4 Experimental Evaluation

4.1 Experimental Methodology

We conducted our experiments on a collection of Pentium-based workstations connected by a lightly-loaded 100Mb/s ethernet. Each workstation has 64 megabytes of memory and runs Solaris 2.5. In our experiments, there is one process of the distributed application per machine. Stable storage is provided by an NFS file server that stores files on a RAID-5 disk array consisting of 6 disks.

For each protocol, we compute our results by averaging the recovery time measured over twenty runs of each of the five applications. For a given application, we guarantee that, independent of the protocol used, failures occur at the same point in the execution of the application: taking advantage of the iterative nature of the applications, we induce process failures after the completion of a pre-determined number of iterations. This ensures that the amount of lost computation that has to be recovered in all three protocols is the same.

4.2 Metrics

For pessimistic and causal protocols, the recovery time (denoted by T_{rec}) for a process comprises of: (1) T_{chk}, the time to restore the state of the failed process from its latest checkpoint stored on the file server,

(2) T_{acq}, the time to retrieve determinants and messages logged during failure-free execution, and (3) $T_{rollfwd}$, the time to roll-forward the execution of the process to its pre-crashed state. For optimistic protocols, on the other hand, in addition to T_{chk} and T_{acq}, the recovery time T_{rec} consists of: (1) T_{replay}, the time to replay messages to the recovering process from the acquired logs, and (2) $T_{rollbck}$, the time overhead for rolling back orphans. Note that T_{acq} is protocol dependent: for pessimistic and optimistic protocols, it is the time to read logs from the file server, while for causal protocols, it is the time to collect messages and determinants from the logs of the remaining processes. In the case of multiple failures, the values of T_{chk}, T_{acq}, $T_{rollfwd}$, T_{replay}, and $T_{rollbck}$ are averaged over the set of concurrently recovering processes.

4.3 Measurements

For all protocols, T_{rec} depends on three parameters.

1. The time t, within the execution interval defined by two successive checkpoint, at which a failure is induced. For all protocols, this parameter affects the amount of lost computation that has to be recovered and the size of the logs that have to be acquired by the recovering process.
2. The number of processes, n. For causal protocols, n may affect T_{acq} because it may chance the set of processes from which a recovering process collects its logs. For optimistic protocols, n may affect $T_{rollbck}$ because it may change the number of orphans.
3. The number of concurrent failures, f. For optimistic protocols, multiple failures may cause a process to rollback multiple times. For sender-based pessimistic and causal protocols, multiple failures may complicate the task of retrieving messages and determinants from other processes.

For optimistic protocols, T_{rec} depends also on the frequency with which volatile logs are flushed to stable storage. In all our experiments, volatile logs are flushed to stable storage every minute. For all protocols, checkpoints are taken six minutes apart.

General Observations Before we proceed to analyze in Figure 1 and Table 2 the effects on T_{rec} of changing the values of t, n, and f, we present a few observation about the behavior of logging protocols that are independent of the specific value of these parameters. We illustrate these observations with the help of Table 1, which shows the result of our experiments when $n = 4$, $f = 1$, and t is chosen half-way between successive checkpoints.

- Messages and determinants that are available in the logs are processed during recovery at a rate higher than during normal execution. This is because, during normal execution a process may have to block waiting for messages, while during recovery these messages can be immediately retrieved from the logs.

  Note that since $T_{rollfwd}$ for pessimistic and causal protocols and $(T_{replay} + T_{rollbck})$ for optimistic protocols dominate the total value of T_{rec}, the reduction in recovery time yielded by processing messages and determinants from logs can be significant. Table 1 illustrates that this reduction is very significant, although it is application-dependent. For communication-intensive applications—such as grid, nbody, life, and p2fox—$T_{rollfwd}$ and $(T_{replay} + T_{rollbck})$ are significantly smaller than 3 minutes, the value of t for this experiment. However, for compute-intensive

Application	**Receiver-based Pessimistic**				**Sender-based Pessimistic**			
	T_{chk} (sec.)	T_{acq} (sec.)	$T_{rollfwd}$ (sec.)	T_{rec} (sec.)	T_{chk} (sec.)	T_{acq} (sec.)	$T_{rollfwd}$ (sec.)	T_{rec} (sec.)
grid	0.31	1.4	30.28	31.99	0.3	3.1	30.5	33.6
nbody	0.29	3.5	78.01	81.8	0.31	4.2	77.5	82.01
gauss	2.61	11.15	207.13	220.89	2.75	14.6	211.01	228.36
life	0.31	0.95	41.1	42.36	0.33	2.1	41.5	43.93
p2fox	1.85	0.8	22.29	24.94	1.8	5.4	22.6	29.8

Application	**Optimistic**					**Causal**			
	T_{chk} (sec.)	T_{acq} (sec.)	T_{replay} (sec.)	$T_{rollbck}$ (sec.)	T_{rec} (sec.)	T_{chk} (sec.)	T_{acq} (sec.)	$T_{rollfwd}$ (sec.)	T_{rec} (sec.)
grid	0.31	1.2	16.8	33.1	51.41	0.36	3.67	30.74	34.77
nbody	0.31	2.91	58.1	37.4	98.72	0.35	4.42	78.1	82.87
gauss	2.64	9.83	197.2	27.6	237.27	2.56	15.7	210.7	228.96
life	0.31	0.8	25.78	40.8	67.69	0.36	2.55	38.97	41.88
p2fox	1.88	0.7	14.7	30.8	48.08	1.87	5.91	21.39	29.17

Table 1: The cost of recovery for $f = 1$, $n = 4$, and failures induced approximately mid-way between successive checkpoints

applications—such as gauss—reading determinants and messages from the logs does not result in any significant speedup.

Note also that in optimistic protocols, any speedup applies only to the portion of the log that is retrieved from stable storage, which in general contains only a prefix of the sequence of the messages and determinants delivered prior to failure. As a result, $(T_{replay} + T_{rollbck})$ for optimistic protocols is typically larger than $T_{rollfwd}$ for receiver-based pessimistic protocols[3].

- Sender-based pessimistic and causal protocols take longer to collect recovery information than receiver-based pessimistic and optimistic protocols. Sender-based pessimistic and causal protocols collect message contents and determinants from operational processes. Although this information is sent concurrently by the operational processes, the recovering process incurs an overhead in merging the received data to create a sequential log used during roll-forward. For receiver-based pessimistic and the optimistic protocols, the logs are already organized sequentially on stable storage. Furthermore, read-ahead, supported by conventional file systems, makes sequential retrieval of the logs efficient.

  Table 1 illustrates that T_{acq} for sender-based pessimistic and causal protocols can be at times six times higher (e.g., in p2fox) than that for receiver-based pessimistic and optimistic protocols. However, since T_{acq} contributes a relatively small fraction of T_{rec}, the impact of this effect on the overall recovery performance of the protocols is minor.

[3]Although for $f = 1$, $(T_{replay} + T_{rollbck})$ is always larger than $T_{rollfwd}$ for all pessimistic and causal protocols, this does not hold for $f > 1$ (see Table 2).