The Cost of Recovery in Message Logging Protocols

Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin

Abstract—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. Our results suggest that applications face a complex trade-off when choosing a message logging protocol for fault tolerance. On the one hand, optimistic protocols can provide fast failure-free execution and good performance during recovery, but are complex to implement and can create orphan processes. On the other hand, orphan-free protocols either risk being slow during recovery, e.g., sender-based pessimistic and causal protocols, or incur a substantial overhead during failure-free execution, e.g., receiver-based pessimistic protocols. To address this trade-off, we propose *hybrid* logging protocols, a new class of orphan-free protocols. We show that hybrid protocols perform within two percent of causal logging during failure-free execution and within two percent of receiver-based logging during recovery.

Index Terms—Distributed computing, fault tolerance, log-based rollback recovery, pessimistic protocols, optimistic protocols, causal protocols, hybrid protocols.

1 Introduction

Message-logging protocols, for example, [2], [3], [6], [8], [11], [12], [19], [20], 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 replaying to it messages in the same order in which 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 [3], [11] 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.
- Optimistic protocols [6], [12], [19], 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. 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 [2], [8], 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 [13] 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, [9], [10], their merits during recovery have been argued mostly qualitatively so far. 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

- *The authors are with the Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712-1188.*
E-mails: {sriram, lorenzo, vin}@cs.utexas.edu.

Manuscript accepted 6 Aug. 1999.
For information on obtaining reprints of this article, please send e-mail to: TKDE@computer.org, and reference IEEECS Log Number 110381.

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.

1041-4347/00/$10.00 © 2000 IEEE

to a consistent state, they require identifying, rolling back, and then rolling 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 (for example, [6], [10], [19]), 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 observed 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 [6] 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 their users could typically afford to invest the resources necessary to mask failures through explicit replication in space [17] 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.

This paper presents the first experimental study of the recovery performance of pessimistic, optimistic, and causal protocols. In the first part of the paper, we study existing protocols by choosing a representative for each style of message logging. For all protocols, we quantify the contribution to the total recovery time of three basic components—checkpoint restore, log retrieval and roll-forward; for optimistic protocols, we also measure the time required to roll back orphan processes. We then compare the relative performance of the three different styles of message logging as a function of the number of concurrent failures, the number of processes in the system, and the time t since the latest checkpoint at which a failure is induced. This study makes two main contributions.

1. It shows that roll-forward time dominates the total recovery time, that roll-forward during recovery can be much faster than normal execution, and that there is a limit to how fast roll-forward can proceed. Given an application and a value for t, it proposes a simple way to compute a lower bound for the roll-forward time, and it identifies the characteristics of a message logging protocol that allow it to approach this lower bound during recovery.
2. It shows that, contrary to our initial intuition, sender-based pessimistic and causal protocols outperform optimistic protocols only when $f = 1$. For $f > 1$, optimistic protocols, although they incur rollbacks, can outperform sender-based pessimistic and causal protocols that are less efficient in supporting fast log retrieval and do not allow fast roll-forward.

Our results suggest that applications face a complex trade off when choosing a message logging protocol for fault tolerance. On the one hand, optimistic protocols can provide fast failure-free execution and good performance during recovery, but are complex to implement and can create orphan processes. On the other hand, orphan-free protocols either risk to be slow during recovery—e.g., sender-based pessimistic and causal protocols—or incur a substantial overhead during failure-free execution—e.g., receiver-based pessimistic protocols. To address this trade off, in the second part of the paper we propose *hybrid* logging protocols, a new class of orphan-free protocols. We repeat our experiments for hybrid protocols and show that they perform within two percent of causal logging during failure-free execution and within two percent of receiver-based logging during recovery.

The rest of the paper is organized as follows. In Section 2, we discuss our implementation of message logging protocols and checkpointing. We describe the application programs used in this study, our experimental methodology, and the metrics for our evaluation in Sections 3.1 to 3.4. Section 3.5 presents an experimental analysis of the recovery costs of the pessimistic, optimistic, and causal logging protocols. Section 4 expands this analysis to study the implications on recovery time of running applications on clusters of nondedicated workstations. Section 5 introduces and evaluates hybrid logging protocols. Finally, Section 6 offers some concluding remarks.

2 Implementation

We measure the cost of recovery in message logging protocols using Egida, an object-oriented toolkit for synthesizing rollback recovery protocols [16]. Egida supports a library of objects that implement a set of functionalities that are at the core of all log-based rollback recovery protocols; different rollback recovery protocols can be implemented by composing objects from this library. Egida is integrated with the MPICH implementation of the Message Passing Interface (MPI) standard [18]. This enables existing MPI applications to take advantage of Egida without any modifications. Using Egida, we implemented a suite of protocols that contain representatives 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* [11]: 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 *send log* kept by the message's sender.[2] This protocol is similar to the one described in [20].

In both of these protocols, the first step of recovering a process p consists of 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 literature, we have implemented the protocol described in [6]. This protocol, in addition to tolerating an arbitrary number of failures and preventing the uncontrolled cascading of rollbacks known as the *domino effect* [19], implements a singularly efficient method for detecting orphans processes. In this protocol, causal dependencies are tracked using vector clocks [14]. 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 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 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 employ two techniques to improve the performance of optimistic protocols. First, we modify the pseudocode presented in [6] 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. This optimization dramatically improves the performance of the protocol during recovery (see Section 3). Second, we fork a child process periodically, every 10 seconds, to flush asynchronously to stable storage the contents of the volatile logs. While the child flushes the logs, the parent process continues to compute, virtually undisturbed.

Causal Logging. We have implemented the Π_{det} family-based message-logging protocol [1]. 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 [2], [8]. Recovery of a failed process proceeds in two phases. In the first phase, the process obtains from the logs of the remaining processes its determinants and 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. In the second phase, the collected data is replayed, restoring the process to its precrash state. To handle multiple concurrent failures, we implemented a protocol that recovers crashed processes without blocking operational processes [7]. In this protocol, the recovering processes elect a leader that collects determinants and messages on behalf of all recovering processes. The leader then forwards the pertinent data to each recovering process.

Egida supports interprocess communication using a modified version of the p4 library [4]. Our version of p4 handles socket errors that occur whenever processes fail and allows a recovering process to establish socket connections with the surviving processes. Finally, the checkpointing module in Egida 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.

3 Experimental Evaluation

3.1 Applications

We use five long-running compute-intensive applications—bt, cg, lu, sp, and mg—from the NPB2.3 benchmark suite developed by NASA's Numerical Aerodynamic Simulation program [5]. These benchmarks are derived from computational fluid dynamics codes; the characteristics of these benchmarks and their communication patterns are shown in Table 1 and Fig. 1, respectively.

3.2 Experimental Setting

We conducted our experiments on a collection of 300 MHz Pentium-II workstations running Solaris 2.7 and connected by a lightly loaded 100 Mb/s Ethernet. Each workstation has 512MB of memory and a 4GB local disk. For each of the benchmark distributed applications, we collect our measures with each workstation hosting one of the application's processes.

We assume that processes can fail by crashing, but that the hardware is reliable. Given this failure model, we implement stable storage using the local disk of each workstation. We discuss the impact of this assumption on our experimental results in Section 3.5.4.

3.3 Metrics

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

2. Some sender-based pessimistic protocols keep both determinants and message contents at the sender [11], [12]. We have not implemented these protocols because they can only tolerate at most two concurrent failures.

TABLE 1
Characteristics of the Benchmarks Used in the Experiments

Application	NPB Specific Info.	Messages/sec.	Message Size (Avg.) (KB)	Exec. Time (sec.)
bt	Class A	6	50.7	2381
cg	Class B	20	60.7	1415
lu	Class A	62	3.7	1064
sp	Class A	22	44.4	1038
mg	Class A	38	46.2	1061

checkpoint, 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 precrashed 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 required to roll 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 shown for the process which takes the longest to recover.

3.4 Experimental Methodology

For all protocols, T_{rec} is determined by the value of three parameters.

1. 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.
2. The number of processes, n. For causal and sender-based optimistic protocols, n may affect T_{acq} because it may change 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 time t, within the execution interval defined by two successive checkpoints, 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.

To compare fairly the recovery performance of the four logging protocols for a given application, t should be the same for all protocols. Furthermore, to compare our results easily across applications, it is convenient to use the same t for all applications. To meet both constraints, we compute the checkpoint interval with respect to an execution in which the applications run with no fault-tolerance. For all applications, checkpoints are taken six minutes apart. Once a t is chosen within this interval, we use the iterative nature of the applications to compute the number of iterations i necessary to run each application, with no underlying fault-tolerance protocol, for t seconds; because the time to complete an iteration is application-dependent, in general, the value of i will change for different applications. Our experiments are conducted by

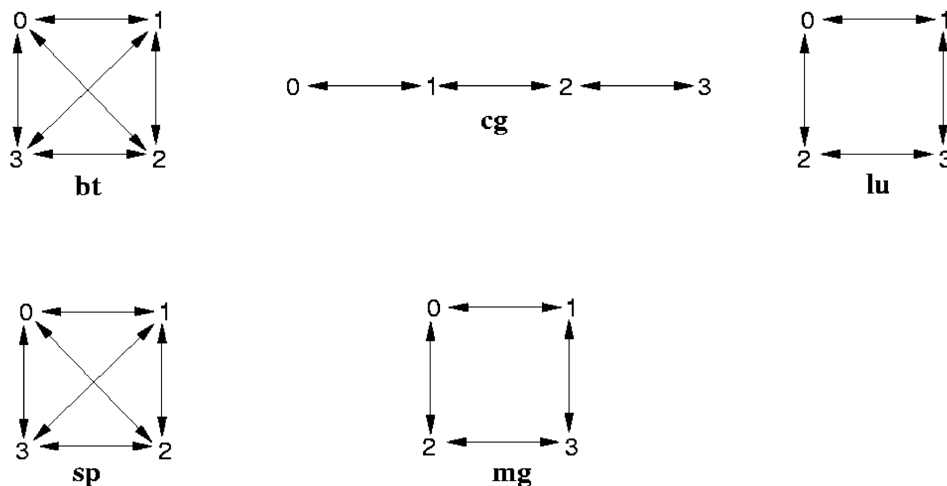

Fig. 1. Communication patterns for our benchmark applications.

TABLE 2
T_{rec} as a Function of f, where $1 \leq f < 4, n = 4$ and $t \approx 3$ min.

Application	f	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.)
bt	1	2.5	2.1	172.4	177.0	2.5	3.6	174.3	180.4
	2	2.6	2.3	172.8	177.7	2.5	2.3	176.9	181.7
	3	2.6	2.1	172.4	177.1	2.5	2.3	180.6	185.4
cg	1	3.7	6.1	139.0	148.8	3.7	24.7	143.3	171.7
	2	3.8	6.4	141.3	151.5	3.7	11.4	161.2	176.3
	3	3.9	6.4	141.4	151.6	3.8	7.7	189.0	200.5
lu	1	0.4	1.1	170.2	171.7	0.4	3.1	170.2	173.7
	2	0.5	1.2	170.3	172.0	0.4	2.1	174.8	177.3
	3	0.5	1.1	170.6	172.2	0.4	2.2	180.0	182.6
sp	1	0.8	5.4	158.0	164.2	0.8	15.3	164.8	180.9
	2	0.8	5.5	157.9	164.2	0.9	10.7	175.2	186.8
	3	0.9	5.4	157.8	164.1	0.9	8.1	186.2	195.2
mg	1	4.1	7.1	165.3	176.5	4.3	17.8	170.4	192.5
	2	4.2	7.7	166.0	177.9	4.4	13.6	188.8	206.8
	3	4.2	7.8	167.4	179.4	4.3	13.2	203.2	220.7

Application	f	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.)
bt	1	2.5	1.9	163.9	14.5	182.8	2.5	3.8	174.6	180.9
	2	2.6	1.8	164.3	14.9	183.6	2.6	8.8	177.8	189.2
	3	2.6	1.7	165.1	15.7	185.1	2.6	6.0	182.2	190.8
cg	1	3.7	5.8	130.4	21.3	161.2	3.7	25.9	143.6	173.2
	2	3.8	5.9	131.2	21.6	162.5	3.8	38.9	161.8	204.5
	3	3.9	6.1	132.7	21.8	164.5	3.8	17.4	187.1	208.3
lu	1	0.4	0.9	158.4	11.1	170.8	0.4	4.1	170.3	174.8
	2	0.4	1.0	159.2	11.2	171.8	0.4	6.4	175.0	181.8
	3	0.5	1.0	160.1	11.4	173.0	0.5	7.2	180.6	188.3
sp	1	0.8	5.1	152.0	16.2	174.1	0.8	16.6	164.7	182.1
	2	0.9	5.1	152.5	16.3	174.8	0.9	36.0	172.4	209.3
	3	0.9	5.2	152.6	16.4	175.1	0.9	24.6	190.1	215.6
mg	1	4.5	6.8	158.3	19.2	188.8	4.4	21.2	171.1	196.7
	2	4.4	6.9	159.2	19.7	190.2	4.4	33.8	190.0	228.2
	3	4.5	7.2	159.4	20.2	191.3	4.4	34.6	205.1	244.1

running each application for its precomputed number of iterations with each of the four logging protocols. We note that, because the overhead of the four protocols varies significantly during failure-free executions, using the same value of t across protocols means that faults are induced at different wall clock times for the different protocols (see Table 3).

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 asynchronously to stable storage every 10 seconds.

For each protocol, we repeat the experiments until the variance for T_{chk}, T_{acq}, $T_{rollfwd}$, T_{replay}, $T_{rollbck}$, and T_{rec} is within one percent of their average values.

3.5 Experimental Results

Before we proceed to analyze the effects on T_{rec} of changing the values of f, t, and n, we present a few observations about the behavior of logging protocols that are independent of the specific values of these parameters. We illustrate these observations with the help of Table 2, which shows the result of our experiments when $1 \leq f \leq 3$, $n = 4$, and t is chosen halfway between successive checkpoints.

- The time to roll forward—$T_{rollfwd}$ for the pessimistic and causal protocols and $(T_{replay} + T_{rollbck})$ for the optimistic protocol—dominates the total recovery time T_{rec}. In most cases, roll forward contributes more than 90 percent of T_{rec} and it never contributes less than 80 percent.
- The sender-based pessimistic and causal protocols collect message contents (and, in the case of causal, determinants) from operational processes and pay the cost of transferring this data over the network. For the receiver-based pessimistic and optimistic protocols, logs are organized sequentially on stable storage; read-ahead, supported by conventional file

TABLE 3
"Wall Clock" Time at which a Failure Is Induced for Each Benchmark Application

Application	Pessimistic		Optimistic	Causal
	Receiver-based (sec.)	Sender-based (sec.)	(sec.)	(sec.)
bt	225.4	197.4	185.2	184.3
cg	362.5	215.6	196.3	194.1
lu	460.3	310.7	197.2	201.5
sp	350.7	240.4	192.8	198.0
mg	390.8	224.3	195.3	202.8

Because of the different fault-tolerance overhead that they impose, the same value of t translates to different wall clock times for the different protocols.

TABLE 4
The Values of t, Communication Time and Basic Computation Time for the Benchmark Applications

Application	t (sec.)	Communication Time (sec.)	Basic Computation Time (sec.)
bt	181.6	8.1	173.5
cg	176.9	37.2	139.7
lu	178.5	9.2	169.3
sp	181.7	21.1	160.6
mg	185.2	20.1	165.1

systems, makes sequential retrieval of the logs efficient.

The result of these effects becomes clear when considering the case of cg for $f = 1$ in Table 2. Using a 100 Mbps Ethernet, with a peak throughput of about 25 Mbps, it takes about 20 seconds to transfer to the recovering process the 65 MB of data it needs during recovery. In contrast, retrieving the same information from local disk requires about five seconds. Even when the difference is not so dramatic, sender-based pessimistic and causal protocols have significantly longer T_{acq} than receiver-based pessimistic and optimistic protocols. Table 2 shows also that for causal and sender-based pessimistic protocols T_{acq} changes with f: we discuss these effects in Section 3.5.1.

- Executing the same sequence of instructions can take significantly less time during roll forward than during normal execution. For instance, consider the performance of cg when $f = 1$. Table 2 and Table 3 show that for this application roll-forward for the causal protocol takes about 17 percent less time than in normal execution, while, for the receiver-based pessimistic protocol, there is a factor of two speedup.

  There are two elements that contribute to a short roll-forward time. First, depending on where they are logged during failure-free execution, it may not be necessary to log messages and determinants again during recovery. Second, if messages and determinants are available from the logs, then it is possible to execute send and receive instructions without incurring the synchronization overhead experienced during the failure-free executions.
- From the previous observation, it follows that, for a given application, there is a lower bound to the time necessary to roll forward. This lower bound is given by the time taken to execute an application assuming that: 1) The application incurs no overhead for fault tolerance and 2) the send and receive operations complete instantaneously. We refer to this lower bound as the *basic computation time* for an application. Table 4 shows the basic computation time for the three benchmark applications, computed as the difference between the time t at which the failure is induced[3] and the *communication time*, i.e., the amount of time spent by the application in performing send and receive operations. This explains why, for applications such as bt and lu that spend little time communicating and exchange small amounts of data, there is little difference in the recovery performance of pessimistic, optimistic, and causal protocols.

  The receiver-based pessimistic protocol comes the closest to implementing assumptions 1) and 2). In this protocol, both the contents of all messages replayed by a recovering process and their corresponding determinants are logged on stable storage and are available during recovery. Hence, send and receive operations incur no blocking

3. Recall from the discussion in Section 3.4 that t refers to an execution with no fault-tolerance overhead.

during roll-forward and, as Table 5 shows, roll-forward imposes no fault-tolerance overhead.

The receiver-based optimistic protocols also provides a good approximation of 1) and 2). However, in this protocol, any speedup applies only to the portion of the log 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 larger than $T_{rollfwd}$ for receiver-based pessimistic protocols. In our implementation, a process asynchronously flushes to stable storage every 10 seconds all messages and determinants held in its volatile log: This puts an upper bound on the extent of roll forward that these protocols cannot run "at full speed." However, as Table 2 shows, $T_{rollbck}$ is always higher that 10 seconds. The cause is the overhead incurred while rolling back orphan processes.

Orphans can be detected only after a recovering process acquires its logs from stable storage. Hence, the rollback and the subsequent roll-forward of an orphan start $T_{chk} + T_{acq}$ from the replay phase of the recovering process. In our applications, this translates in turn to an equivalent delay that a recovering process incurs after having replayed its logs, while it waits for the orphans to finish the replay of their logs. This delay occurs only if the recovering process waits to receive a message that is to be sent by an orphan process that has not completed the replay of its log—a scenario that occurs in all the applications used in this study. Since the time required for replaying messages from the logs is nearly the same for a recovering process and an orphan the value of $T_{rollbck}$ is approximately equal to $10 + T_{chk} + T_{acq}$.

The sender-based pessimistic and causal protocols do not perform as well. In these protocols, determinants are available during recovery, but messages maintained in the volatile logs of faulty processes are lost and must be regenerated and once again logged. This has two consequences. First, reconstructing the logs involves performing memory-to-memory copies; in our experimental setting, this involves an overhead of about $100\mu s$ per 1 KB data in addition to the cost of memory allocation. Table 5 shows that, for cg, where the size volatile log of sent messages is about 65 MB, the overhead of reconstructing the log is about 7 seconds for both the causal and sender-based pessimistic protocols. Second, when $f > 1$, some of the messages needed by a recovering process may not be immediately available for replay. In this case, the blocking incurred by send and receive operations during failure-free executions can occur again during recovery. We discuss this effect in greater detail in Section 3.5.1.

- In optimistic protocols, overlapping the roll-forward of recovering processes with the identification, rollback and roll-forward of orphans significantly reduces the cost of recovery.

  For instance, Table 2 shows that, in the case of cg, the value of T_{rec} for $f = 1$ for the optimized protocol is approximately 160 seconds, compared with the value of 288.6 seconds we obtained by running the protocol without the optimization. Similar results hold for the other applications.
- Since the size of the process state saved in checkpoints is independent of the message-logging protocols, for a given application T_{chk} is virtually constant across all protocols, as we expected.

3.5.1 Changing the Number of Concurrent Failures

Receiver-based pessimistic. The performance of this protocol does not depend on the value of f because each recovering process can retrieve from stable storage all the data necessary to complete roll-forward in the minimum time possible.

Optimistic. The performance of the optimistic protocol is virtually unaffected by a variation in f. In our implementation, all messages but those received by a faulty process in the last 10 seconds are available at the beginning of the roll-forward phase. Although unavailable messages are regenerated and replayed at the slower speed of normal execution, the messages available on stable storage can be processed quickly, while, in parallel, orphans are first rolled back and then rolled forward.

As a result, when $f = 1$, the optimistic protocol performs only slightly worse than causal and pessimistic protocols, both sender and receiver based. When $f > 1$, however, the optimistic protocol can perform substantially better than

TABLE 5
Overhead Imposed by Protocols during the Roll-Forward Phase of Recovery

Application	Pessimistic		Optimistic	Causal
	Receiver-based (sec.)	Sender-based (sec.)	(sec.)	(sec.)
bt	0	1.2	0.2	1.3
cg	0	6.9	0.4	7.2
lu	0	0.8	0.1	0.9
sp	0	5.4	0.2	5.5
mg	0	6.4	0.3	6.6

both the sender-based pessimistic and the causal protocols (see Fig. 2). This result surprised us: Before starting this study, we believed that pessimistic and causal protocols, which never force rollbacks, would always recover faster than optimistic protocols.

Causal and sender-based pessimistic. Changing the number f of concurrent failures significantly affects the degree to which these protocols approximate the conditions under which roll-forward time is minimized.

- When $f = 1$, all the determinants and messages that the faulty process needs to replay are available at the beginning of the roll-forward phase. Hence, during roll-forward, there is no blocking on send and receive operations. $T_{rollfwd}$ should therefore exceed the basic computation time only because of the overhead of refilling the volatile send log of the faulty process. A comparison of the values reported in Table 2 and Table 5 shows that our experimental results support this interpretation.
- When $f > 1$, the determinants of all messages are still available at the beginning of the roll-forward phase. However, this does not hold for to the contents of these messages. In particular, all messages originally sent by any of the concurrently failed processes are temporarily lost. As a result, a recovering process will stop rolling forward whenever it encounters one of these missing messages, waiting for the sender to recover to the point at which the message is regenerated and resent. We call this effect *stop-and-go*. Whenever stop-and-go occurs, the roll-forward phase slows down to the speed of normal execution. Fig. 2 shows that, as f increases and more messages are temporarily lost, stop-and-go has an increasingly adverse effect on T_{rec} for cg. A similar effect holds also for the other applications, but it is less significant because bt and lu have a much smaller communication time than cg (see Table 2 and Table 4).
- In causal protocols, T_{acq} increases significantly when there are multiple concurrent failures, as the overhead of the algorithm [7] used for acquiring messages and determinants becomes higher when $f > 1$. There are two elements of the algorithm that contribute to this behavior. First, the recovering processes need to elect a recovery leader. Second, the recovery information is first gathered by the recovery leader, which then forwards it to each recovering process. We expect that, with a different recovery protocol that allows each process to collect its recovery information independently [15], the values of T_{acq} in the causal and sender-based pessimistic protocols would be similar.

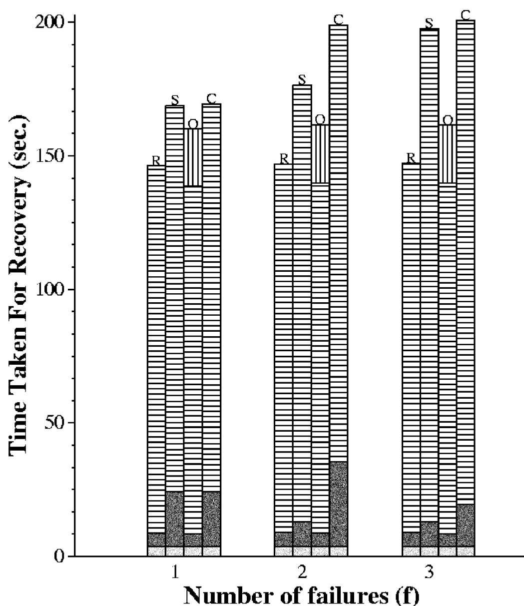

Fig. 2. T_{rec} as a function of f, where $1 \leq f \leq 3$, for cg, $n = 4$. For each value of f, we show four bars: receiver-based pessimistis (R), sender-based pessimistic (S), optimistic (O), and causal (C). For legend see Fig. 3.

3.5.2 Changing the Time of Failure

Fig. 3a shows the effect of varying t on T_{rec} for lu. We make two observations. First, as t increases, so do both T_{acq} and, more significantly, $T_{rollfwd}$ and $(T_{replay} + T_{rollbck})$. This is not surprising because, for higher values of t, more messages and determinants need to be acquired and processed. Second, since $T_{rollbck}$ depends only on the frequency at which the logs are flushed to stable storage, its value does not change with t. Consequently, the contribution of $T_{rollbck}$ to T_{rec} is proportionally reduced.

3.5.3 Changing the Number of Processes

We assess the effect on T_{rec} of varying n for lu, an application in which each process communicates with every other process. We expected to observe that increasing n would increase T_{acq} for causal and sender-based pessimistic protocols, as well as $T_{rollbck}$ for optimistic protocols. Fig. 3b indicates that these effects, although present, are insignificant.

We assess the effect on T_{rec} of varying n for lu. For the causal and sender-based pessimistic protocols, we expected to observe that increasing n would increase T_{acq}. Fig. 3b indicates that this effect is present, but small. For the optimistic protocol, increasing n can increase the number of processes that have to rollback whenever there is a failure. Because orphans roll back and roll forward concurrently, however, this has practically no effect on T_{rec}.

3.5.4 Changing the Failure Model

If we weaken the failure model to include hardware failures, then local disks are no longer valid implementations of stable storage. Stable storage can instead be implemented on a highly available, remote file system, such as NFS. To understand how the new implementation of stable storage affects recovery time, we restart the crashed process on a different machine and measure T_{rec}. Comparing the results of our new experiments, shown in Table 6, with the measures reported in Table 2, we see that

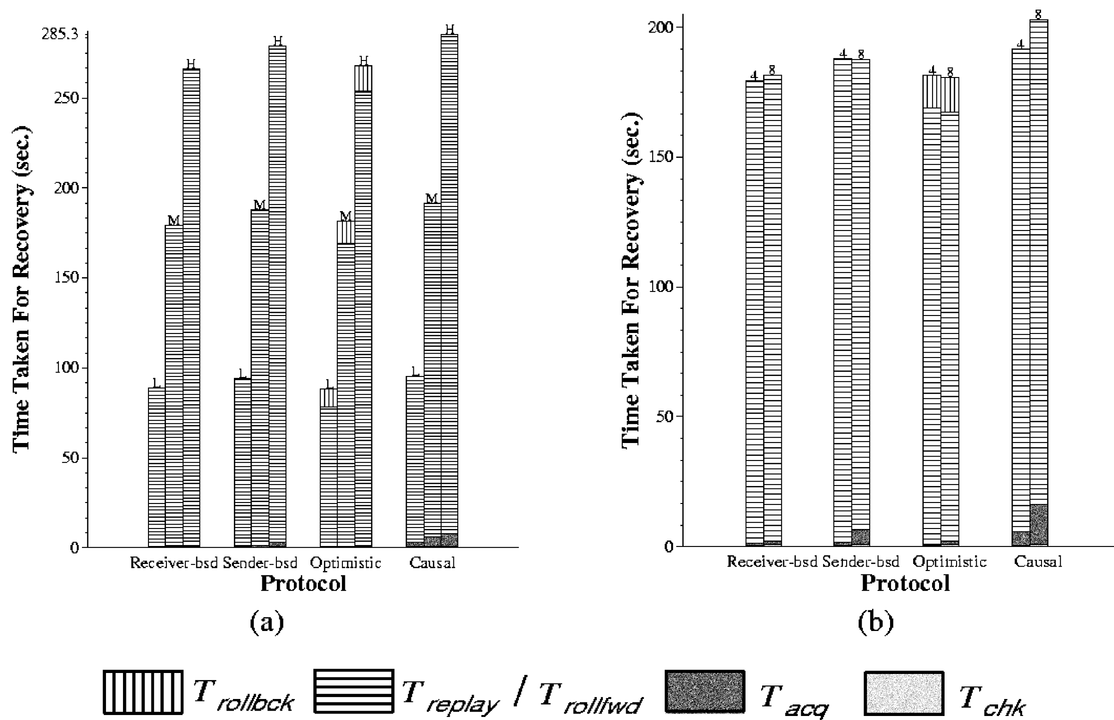

Fig. 3 (a) Effect on the cost of recovery for lu $(n = 4, f = 1)$ of inducing failure after approximately a fourth (L), half (M), and three-fourth (H) of the interval between successive checkpoints. (b) Cost of recovery for lu with four and eight processes, $f = 1,$ and $t \approx 3$ min.

the new implementation of stable storage has two effects on the overall recovery time:

1. For all protocols, restoring a checkpoint involves transferring data from a remote file server. The time required to complete this transfer depends both on the size of the checkpoint file and on the load on the file server. The latter, in turn, depends on the value of f: As multiple recovering processes try to retrieve their checkpoints and logs from the file server concurrently, their requests end up being sequentialized at the network interface of the file server, adding to the retrieval time.
2. The cost of acquiring the logs from stable storage increases for all but causal protocols. In receiver-based pessimistic and optimistic protocols, a recovering process must now retrieve, from a remote server, messages and determinants that used to be available on its local disk, with performance implications similar to those discussed above for checkpoints. Note that sender-based pessimistic protocols are less affected because the new failure model requires changing only the location where determinants are stored; messages are stored, as before, in the volatile memory of the sender.

4 Recovery Performance in the Presence of Load Imbalance across Workstations

Our benchmark applications partition the overall computational task into several roughly equally sized iterations. Each application process executes one such iteration and exchanges messages with other processes to share intermediate results and to synchronize its execution with them. In Section 3, we reported the results obtained executing the application processes on a unloaded homogeneous cluster of workstations. In such an environment, the execution of the processes tends to proceed in lock-step; the receive and send operations performed by communicating processes are roughly synchronized. Hence, even though the processes executing the benchmark applications exchange a large number of messages, they incur a relatively small blocking overhead (see Table 1 and Table 4).

In reality, however, application processes may execute on workstations with different and varying workloads. In these environments, processes may take different amounts of time to execute each iteration of the application. The resulting asynchrony in process execution may increase the overheads due to blocking. To study this effect, we developed the following two-step methodology. First, for each of the benchmark applications, we use the trace obtained by executing the application on the a unloaded homogeneous cluster of workstations to derive a new execution trace. Using this new trace, we define a synthetic application that simulates the behavior of the original application on a cluster of workstations with a varying and uneven load. Second, we measure the cost of recovery of all protocols for the synthetic application. The first step involves the following tasks:

TABLE 6
T_{rec} as a function of f, where $1 \leq f < 4, n = 4$ and $t \approx 3$ min., with Stable Storage Implemented Using the Disk of an NFS Server

Application	f	Receiver-based Pessimistic T_{chk} (sec.)	T_{acq} (sec.)	$T_{rollfwd}$ (sec.)	T_{rec} (sec.)	Sender-based Pessimistic T_{chk} (sec.)	T_{acq} (sec.)	$T_{rollfwd}$ (sec.)	T_{rec} (sec.)
bt	1	10.1	3.8	173.0	186.9	10.2	3.7	174.7	188.6
	2	16.6	4.7	172.9	194.2	15.4	3.0	176.1	194.5
	3	25.3	7.1	172.6	205.0	23.9	2.6	180.5	206.7
cg	1	16.2	28.3	138.8	183.3	16.4	26.1	142.7	185.2
	2	24.8	35.2	141.6	201.6	25.7	11.9	162.8	200.4
	3	40.6	42.1	141.8	224.5	40.1	8.8	189.9	238.8
lu	1	1.9	3.3	170.6	175.8	1.9	3.4	171.2	176.5
	2	2.1	4.4	170.8	177.3	2.0	2.5	175.1	179.6
	3	2.9	5.8	170.9	179.6	2.7	2.6	180.7	186.0
sp	1	3.4	17.8	159.2	180.4	3.3	16.0	163.0	182.3
	2	4.6	22.3	159.7	213.2	4.2	12.1	175.6	191.9
	3	7.2	26.2	159.6	193.0	5.9	8.3	185.4	200.6
mg	1	21.8	20.2	165.4	207.4	21.1	18.1	170.6	209.8
	2	32.4	26.3	165.7	224.4	33.7	14.4	187.5	235.8
	3	51.3	34.3	165.6	251.2	52.6	14.1	204.3	269.9

Application	f	Optimistic T_{chk} (sec.)	T_{acq} (sec.)	T_{replay} (sec.)	$T_{rollbck}$ (sec.)	T_{rec} (sec.)	Causal T_{chk} (sec.)	T_{acq} (sec.)	$T_{rollfwd}$ (sec.)	T_{rec} (sec.)
bt	1	10.3	3.5	164.2	16.2	194.2	10.2	4.0	174.7	188.9
	2	16.1	4.4	165.3	16.3	202.1	15.8	8.3	177.7	201.8
	3	24.2	6.6	165.4	16.6	212.8	24.7	6.2	181.4	212.3
cg	1	16.8	24.2	131.6	22.1	194.7	16.5	27.3	143.8	187.6
	2	25.4	33.8	131.5	23.2	213.9	24.9	40.1	161.9	226.9
	3	41.1	39.3	132.1	23.4	235.9	40.5	14.2	188.9	243.6
lu	1	1.8	2.8	159.2	11.8	175.6	1.9	4.5	170.8	177.2
	2	2.2	3.8	161.1	12.1	179.2	2.0	6.6	176.2	184.8
	3	3.1	5.1	160.8	12.4	181.4	2.7	7.1	181.2	191.0
sp	1	3.5	16.3	153.1	18.4	191.3	3.3	17.6	164.6	185.5
	2	4.3	21.4	153.4	18.1	197.2	4.4	36.3	172.8	213.2
	3	7.3	25.1	152.9	18.6	203.9	7.1	25.4	190.8	223.5
mg	1	22.1	19.1	159.4	20.3	220.9	22.3	20.3	170.0	212.6
	2	34.3	25.7	160.1	21.2	241.3	34.2	33.3	187.7	255.2
	3	51.8	33.2	160.3	21.6	266.9	52.3	34.5	206.1	292.9

1. For each application process, we measure: 1) the time at which each loop iteration begins, 2) the times at which the process sends or receives messages during that loop iteration, and 3) the time at which the loop iteration completes. From this application trace, we determine the mean μ and the standard deviation σ for the time spent executing each loop iteration.
2. We model the time to execute a loop iteration on workstations with a varying and uneven workload as an *exponential random variable* with mean μ. Using this exponential distribution, we derive a sequence of loop iteration times in the range $[\mu - \sigma, \mu + C * \mu]$, where C is a constant. Restricting the range of loop execution times eliminates from consideration the extreme cases of the exponential distribution.
3. We use the resulting sequence of loop iteration times to scale appropriately the time offsets—with respect to the beginning of each loop iteration—at which the process executes sends and receives.

This method does not alter the frequency of communication or the amount of information exchanged between application processes; it simply introduces some variation in the time required to execute each loop iteration.

Table 7 shows the execution time t, the communication time, and the basic computation time for each of the synthetic applications. Before inducing a failure, we run a synthetic application for the same number of iteration, shown in Table 3, as its original counterpart. Since, on average, the time to complete an iteration is larger for the synthetic applications, the values of t in Table 7, are higher than the corresponding values in Table 4.

Table 7 also illustrates that, when loop iteration times are selected from the range $[\mu - \sigma, 2\mu]$, the communication times for the synthetic bt and lu applications are about an order of magnitude larger than the corresponding values for the original applications, whereas the computation times are only about 50 percent larger (see Table 4). For the synthetic cg application, on the other hand, the percentage increase in both the communication and computation time are roughly the same. These results indicate that, when

TABLE 7
The Values of t, Communication Time and Basic Computation Time for the Synthetic Applications

Synthetic Application	$[\mu - \sigma, 1.5 * \mu]$		
	t (sec.)	Communication Time (sec.)	Basic Computation Time (sec.)
bt	250.6	38.3	212.3
cg	274.5	55.9	218.6
lu	288.6	63.4	225.2
sp	275.6	50.0	225.6
mg	285.9	48.9	237.0

Synthetic Application	$[\mu - \sigma, 2 * \mu]$		
	t (sec.)	Communication Time (sec.)	Basic Computation Time (sec.)
bt	317.3	72.0	245.3
cg	362.7	82.0	280.7
lu	342.3	87.7	254.6
sp	341.5	78.5	263.0
mg	355.5	77.7	277.8

TABLE 8
Recovery Performance of Causal Protocol for Synthetic Applications ($1 \leq f < 4$ and $n = 4$)

Synthetic Application	f	$[\mu - \sigma, 1.5 * \mu]$ $T_{rollfwd}$ (sec.)	$[\mu - \sigma, 2 * \mu]$ $T_{rollfwd}$ (sec.)
bt	1	213.7	244.5
	2	229.8	297.5
	3	242.9	316.6
cg	1	215.7	277.7
	2	259.1	338.9
	3	277.6	350.8
lu	1	226.1	256.1
	2	282.5	313.4
	3	296.1	336.8
sp	1	229.1	267.1
	2	244.1	293.4
	3	264.1	325.6
mg	1	241.4	279.9
	2	269.6	323.6
	3	289.5	351.4

different processes execute loop iterations of unequal length, the blocking incurred executing receive and send operations increases substantially.

This increase does not affect $T_{rollfwd}$ for the receiver-based pessimistic protocol. In this protocol, messages and determinants are available from stable storage. As a result, for a given application for this protocol, $T_{rollfwd}$ is virtually identical to the application's basic computation time. A similar argument also applies to the optimistic protocol. Since the volatile logs are flushed asynchronously to stable storage every 10 seconds, any increased blocking during failure-free execution can affect at most the last 10 seconds of roll forward: The influence on $T_{rollfwd}$ is therefore only marginal. For the sender-based pessimistic and causal protocols, on the other hand, we have seen in Section 3.5.1 that, for $f > 1$, less messages are available at the beginning of the roll-forward phase. This causes what we called the stop-and-go effect: Roll forward proceeds at the speed of failure-free execution and $T_{rollfwd}$ departs from the basic computation time.

The higher blocking experienced by synthetic applications can magnify the stop-and-go effect: While, for the original bt and lu applications, $T_{rollfwd}$ increased by about five percent (see Table 2) from $f = 1$ to $f = 3$, Table 8 shows that for their synthetic counterparts the difference grows to about 30 percent. Similar results also hold for the sender-based pessimistic protocol.

5 Hybrid Protocols

Our study suggests that applications face a complex trade-off when choosing a message logging protocol for fault tolerance. Optimistic protocols can combine fast failure-free execution with good performance during recovery, but at the cost of renouncing fault containment and of facing a complex implementation task. Orphan-free protocols, however, either risk being relatively slow during recovery, like sender-based pessimistic and causal protocols, or incur a substantial overhead during failure-free execution, like receiver-based pessimistic protocols (see Table 9).

To address this trade-off, we introduce a new class of *hybrid* protocols. The objective of hybrid protocols is to maintain the failure-free performance of sender-based

TABLE 9
Failure-Free Overhead Imposed by the Various Logging Protocols

Application	Pessimistic				Optimistic		Causal	
	Receiver-based		Sender-based					
	Exec. Time (sec.)	Overhead	Exec. Time (sec.)	Overhead	Exec. Time (sec.)	Overhead	Exec. Time (sec.)	Overhead
bt	4208	76.7%	2617	9.9%	2480	4.2%	2491	4.6%
cg	4051	186.3%	1810	27.9%	1567	10.7%	1580	11.7%
lu	3947	271.0%	2203	107.0%	1209	13.6%	1237	16.3%
sp	2697	159.8%	1580	52.3%	1231	18.6%	1208	16.4%
mg	3943	271.6%	2000	88.5%	1260	18.8%	1245	17.3%

TABLE 10
T_{rec} in Hybrid Protocols as a Function of f, where $1 \leq f < 4$, $n = 4$, and $t \approx 3$ min.

Application	f	Hybrid Sender-based Pessimistic				Hybrid Causal			
		T_{chk} (sec.)	T_{acq} (sec.)	$T_{rollfwd}$ (sec.)	T_{rec} (sec.)	T_{chk} (sec.)	T_{acq} (sec.)	$T_{rollfwd}$ (sec.)	T_{rec} (sec.)
bt	1	2.5	2.6	173.5	178.6	2.5	2.7	173.4	178.6
	2	2.6	2.8	174.8	180.2	2.6	3.2	174.3	180.1
	3	2.6	2.5	175.6	180.7	2.6	2.9	176.9	182.4
cg	1	3.7	6.5	140.6	150.8	3.8	6.8	140.8	151.4
	2	3.7	6.7	141.3	151.7	3.8	7.4	142.1	153.4
	3	3.9	6.9	143.2	154.0	3.8	7.1	142.8	153.7
lu	1	0.5	1.4	170.1	172.0	0.4	1.5	170.3	172.2
	2	0.6	1.7	171.3	173.6	0.6	1.8	171.5	173.9
	3	0.5	1.6	173.2	175.3	0.4	1.9	172.9	175.2
sp	1	0.9	5.8	159.8	166.5	0.8	5.6	161.9	168.3
	2	0.9	6.1	160.1	167.1	0.9	6.4	163.6	170.9
	3	1.0	6.3	161.2	168.5	0.9	6.2	165.7	172.8
mg	1	4.2	7.3	166.2	177.7	4.5	7.5	167.8	179.8
	2	4.3	7.5	169.3	181.1	4.2	7.9	169.3	181.4
	3	4.5	7.8	172.2	184.5	4.6	8.1	171.4	184.1

protocols while approaching the performance of receiver-based protocols during recovery. In hybrid protocols, messages are logged both at the sender *and* at the receiver. The sender synchronously logs messages in its volatile memory; the receiver asynchronously logs messages to stable storage. Since logging on stable storage is asynchronous, performance during failure-free executions is virtually identical to that of sender-based protocols. However, recovery is much faster. The log on stable storage contains a prefix of the messages that are to be replayed during recovery; while this prefix is replayed, the stop-and-go effect cannot occur. Any missing message is either available in the volatile memory of other operational processes or it has to be regenerated during recovery if the sender has failed. In either case, no correct process ever becomes an orphan and the recovering process can roll forward using the messages on stable storage while in parallel acquires the missing messages.

We have implemented and evaluated hybrid versions of sender-based pessimistic and causal protocols. Table 10 shows that hybrid logging dramatically reduces the recovery cost of the sender-based pessimistic and causal protocols, which are now within two percent of the receiver-based protocol, even when $f > 1$ (see Table 2 and Table 10). At the same time, Table 11 shows that when hybrid logging is used in place of sender-based logging, the failure-free execution time increases by at most two percent. In practice, hybrid causal protocols are more desirable because causal logging imposes significantly less overhead during failure-free executions than sender-based pessimistic protocols (see Table 9).

6 Concluding Remarks

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. Message logging protocols have been proposed as a promising technique for achieving fault-tolerance with little overhead. The relative overhead that these protocols impose during failure-free executions is well-understood. This is not the case, however, for their performance during recovery, which has so far been argued mostly qualitatively.

In this paper, we presented the first experimental evaluation of the performance of message logging protocols during recovery. We discovered that roll-forward time dominates the total recovery time for all the protocols and that roll-forward can proceed much faster than normal execution. We derived a lower bound for the roll-forward time and identified the characteristics that allow a message logging protocol to approach this lower bound during recovery. We showed that receiver-based pessimistic

TABLE 11
Increase in the Failure-free Overhead Imposed by Hybrid Protocols when Compared to Their Traditional Counterparts

Application	Hybrid Sender-based Pessimistic	Hybrid Causal
bt	1.2%	1.6%
cg	1.1%	1.9%
lu	0.9%	1.6%
sp	1.3%	1.8%
mg	1.2%	1.7%

protocol always achieves this lower bound and, hence, has the best recovery performance. We also reported that if a single failure is to be tolerated, pessimistic and causal protocols perform best, because they avoid roll-backs 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).

The above results identify two design principles. First, it is a bad idea to rely on other processes to provide the messages that have to be redelivered during recovery. Protocols that do not operate according to this principle, such as sender-based pessimistic and causal protocols, incur significantly higher roll-forward costs. Second, it is a good idea to design optimistic protocols so that orphan processes can be identified, rolled back, and rolled forward in parallel with the roll-forward of the recovering processes. This principle is not surprising. However, it is largely ignored by current optimistic protocols, which instead concentrate on minimizing performance metrics that can be quantified analytically, such as the number of rounds needed to detect orphans. Although these metrics are interesting, they focus on aspects of the recovery protocol that affect T_{rec} only marginally.

Finally, our analysis suggests that applications face a trade-off when choosing a message logging protocol for fault tolerance. Optimistic protocols can combine fast failure-free execution with good performance during recovery, but at the cost of renouncing fault containment and of facing a complex implementation task. Orphan-free protocols, however, either risk to be slow during recovery, sender based pessimistic and causal protocols, or incur a substantial overhead during failure-free execution, receiver-based pessimistic protocols. We proposed a new class of orphan-free protocols, referred to as hybrid protocols, that break this trade-off. We showed that hybrid protocols perform within two percent of causal logging during failure-free execution and within two percent of receiver-based logging during recovery.

ACKNOWLEDGMENTS

We would like to thank Mukesh Singhal for his encouragement. This research was supported in part by the U.S. National Science Foudation (CAREER Award Nos. CCR-9733842 and CCR-9624757 and Research Infrastructure Grant CDA-9624082), DARPA/SPAWAR (Grant No. N66001-98-8911), IBM, and Intel.

REFERENCES

[1] L. Alvisi and K. Marzullo, "Tradeoffs in Implementing Optimal Message Logging Protocols," *Proc. Fifth ACM Symp. Principles of Distributed Computing,* pp. 58-67, June 1996.

[2] L. Alvisi and K. Marzullo, "Message Logging: Pessimistic, Optimistic, Causal, and Optimal," *IEEE Trans. Software Eng.,* vol. 24, no. 2, pp. 149-159, Feb. 1998.

[3] A. Borg, J. Baumbach, and S. Glazer, "A Message System Supporting Fault Tolerance," *Proc. Symp. ACM SIGOPS Operating Systems Principles,* pp. 90-99, Oct. 1983.

[4] R. Butler and E. Lusk, "Monitors, Message, and Clusters: The p4 Parallel Programming System," *Parallel Computing,* vol. 20, pp. 547-564, Apr. 1994.

[5] "NAS Parallel Benchmarks,"NASA Ames Research Center, http://science.nas.nasa.gov/Software/NPB/, 1997.

[6] O.P. Damani and V.K. Garg, "How to Recover Efficiently and Asynchronously when Optimism Fails," *Proc. 16th Int'l Conf. Distributed Computing Systems,* pp. 108-115, 1996.

[7] E.N. Elnozahy, "On the Relevance of Communication Costs of Rollback-Recovery Protocols," *Proc. 14th Ann. ACM Symp. Principles of Distributed Computing,* pp. 74-79, Aug. 1995.

[8] E.N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit," *IEEE Trans. Computers,* vol. 41, no. 5, pp. 526-531, May 1992.

[9] E.N. Elnozahy and W. Zwaenepoel, "On the Use and Implementation of Message Logging, *Digest of Papers: 24th Ann. Int'l Symp. Fault-Tolerant Computing,* June 1994.

[10] D.B. Johnson, "Distributed System Fault Tolerance Using Message Logging and Checkpointing," PhD thesis, report no. COMPTR89-101, Rice Univ., Dec. 1989.

[11] D.B. Johnson and W. Zwaenepoel, "Sender-Based Message Logging," *Digest of Papers: 17th Ann. Int'l Symp. Fault-Tolerant Computing,* June 1987.

[12] T.Y. Juang and S. Venkatesan, "Crash Recovery with Little Overhead," *Proc. 11th Int'l Conf. Distributed Computing Systems,* pp. 454-461, June 1987.

[13] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, July 1978.

[14] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms,* M. Cosnard et. al., eds., Elsevir Science Publishers B.V., 1989.

[15] J.R. Mitchell and V.K. Garg, "A Non-Blocking Recovery Algorithm for Causal Message Logging," *Proc. 17th Symp. Reliable Distributed Systems,* West Lafayette, Ind., pp. 3-9, Oct. 1998.

[16] S. Rao, L. Alvisi, and H.M. Vin, "Egida: An Extensible Toolkit for Low-Overhead Fault-Tolerance," *Proc. IEEE Fault-Tolerant Computing Symp. FTCS-29,* Madison, Wis., pp. 48-55, June 1999.

[17] F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial, *Computing Surveys,* vol. 22, no. 3, pp. 299-319, Sep. 1990.

[18] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, "Scientific and Engineering Computation Series," *MPI: The Complete Reference,* Cambridge, Mass.: MIT Press, 1996.

[19] R.B. Strom and S. Yemeni, "Optimistic Recovery in Distributed Systems," *Proc. ACM Trans. Computer Systems,* vol. 3, no. 3, pp. 204-226, Apr. 1985.

[20] R.E. Strom, D.F. Bacon, and S.A. Yemini, "Volatile Logging in n-Fault-Tolerant Distributed Systems," *Proc. Third Ann. Int'l Symp. Fault-Tolerant Computing,* pp. 44-49, 1988.

Sriram Rao received his PhD in computer science from the University of Texas at Austin in 1999. He also received his MS and BS (with high honors) from the University of Texas at Austin in 1994 and 1992, respectively. He was a recipient of the Microelectronics and Computer Development (MCD) fellowship awarded by the University of Texas, Department of Computer Sciences. His research interests include fault tolerance, distributed systems, and multimedia systems. He is currently employed by Inktomi Corporation.

Lorenzo Alvisi received his PhD in computer science from Cornell University in 1996. He had previously received an MS in computer science from Cornell, and a Laurea in Physics from the University of Bologna, Italy. He is currently an assistant professor and faculty fellow in the Department of Computer Sciences at the University of Texas at Austin, where he co-founded the Laboratory for Experimental Software Systems (UT LESS). He has received several awards, including the U.S. National Science Foundation CAREER award. Dr. Alvisi's research interests include distributed systems and fault tolerance.

Harrick M. Vin received his PhD in computer science from the University of California at San Diego in 1993. He is currently an associate professor and a faculty fellow of computer sciences, and the director of the Distributed Multimedia Computing Laboratory at the University of Texas at Austin. His research interests are in the areas of multimedia systems, integrated services networks, fault tolerance, and distributed systems. He has coauthored more than 75 papers in leading journals and conferences. He has been a recipient of several awards, including the U.S. National Science Foundation CAREER award, IBM Faculty Development Award, AT&T Foundation Award, IBM Doctoral Fellowship, NCR Innovation Award, and the San Diego Supercomputer Center Creative Computing Award.