Egida: A Toolkit For Low-overhead Fault-tolerance

by

Sriram S. Rao, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 1999

Egida: A Toolkit For Low-overhead Fault-tolerance

Publication No. ____________________

Sriram S. Rao, Ph.D.
The University of Texas at Austin, 1999

Supervisors: Lorenzo Alvisi and Harrick M. Vin

Log-based rollback recovery protocols—such as logging and checkpointing—are an attractive solution for building non-critical applications that can tolerate crash failures. Surprisingly though, very few of these protocols are being used in practice to build reliable applications. We believe that this is because of two reasons. First, the few existing implementations of rollback-recovery protocols either support only a single protocol or require a developer to implement each protocol from scratch. Second, the performance of rollback-recovery protocols in practice is not well understood.

To address these issues, we develop *Egida*, an extensible toolkit for rollback-recovery. To design Egida, we develop a single unifying framework based on handling non-deterministic events in process execution. This framework allows us to characterize rollback-recovery protocols as event-driven programs that differ in their response to a same set of "relevant" events and to identify a set of functionalities that are at the core of all these protocols. To express rollback recovery protocols in terms of the relevant events, we have developed a simple specification language. From a specification, Egida can synthesize the corresponding protocol implementation by gluing together appropriate implementations of core mod-

ules from an available library. Egida is extensible and facilitates rapid implementation of rollback-recovery protocols with minimal programming effort.

We use Egida to investigate three fundamental performance issues. First, we study the recovery performance of message-logging protocols and discover that existing protocols do not at the same time guarantee low-overhead during failure-free execution, fast recovery, and fault-containment. We develop a new class of *hybrid* protocols that for the first time provide these three benefits simultaneously. Second, we evaluate the performance of communication-induced checkpointing (CIC) protocols and show that CIC protocols are much less effective in practice than previously believed. Third, we study the overhead imposed by message-logging protocols when processes interact with the file system. Our results show that existing protocols impose a substantially high overhead when processes also communicate by reading and writing shared files. We develop a novel file-sharing protocol that reduces this overhead dramatically.

Contents

Chapter 1

Introduction

The computing revolution of the 90's has seen clusters of interconnected desktop workstations replace mainframe computers as the basic unit of the computing infrastructure. This environment has facilitated the development of a large class of distributed applications in a variety of disciplines such as Cosmology (e.g., simulating collision of galaxies [27]), Chemistry (e.g., determining electronic structure of molecules [58]), Physics (e.g., computational fluid dynamics codes [17]), Computer Security (e.g., cracking RSA cryptosystems [47]), and Meteorology (e.g., cloud detection [46]). All of these applications are long-running, computationally intensive tasks structured around groups of communicating processes that execute on different machines in a workstation cluster.

In building these emerging applications, one is forced to consider the failure of one or more processes involved in the computation. This is because the failure of even a single process can cause the application to terminate prematurely or to produce incorrect results. Although failures may be trivially handled by re-starting the application from the initial state, re-starting an application after it has executed from several hours to a few days will only waste valuable user time.

For providing fault-tolerance to these emerging applications, rollback-recovery protocols such as checkpointing and message-logging are a promising approach. These protocols require substantially fewer dedicated resources when compared to alternate techniques based on active-replication [73]. Rollback-recovery protocols are also relatively simple to use. They provide the same send/receive programming model which is commonly used for structuring message-passing distributed applications. These protocols are well-studied. A recent survey paper on rollback-recovery [23] contains over 300 references to the literature on these protocols. Surprisingly, in spite of such a large body of work, rollback-recovery protocols are rarely used for building reliable distributed applications. We believe that this is because of two reasons:

1. Effectively integrating a rollback-recovery protocol with an application is complex. The few implementations of rollback recovery protocols that are available either support only a single protocol [16, 25, 34, 48, 60, 77] and thereby limit adapting protocols to the specific needs of different applications or require a developer to implement each protocol underneath the basic send and receive interface [57]. Consequently, application developers in addition to worrying about the complexity of building their application, are also forced to understand the subtleties of distributed fault-tolerance.

2. The performance of rollback-recovery protocols in practice is not well understood. Proponents of new protocols have typically focussed on the algorithmic aspects of protocol design, while providing little experimental evaluation that demonstrates the merits of their new approach when compared to that of their predecessors. As a result, application developers have little guidance in choosing from amongst the large class of rollback-recovery protocols the one that best matches the characteristics of their particular application.

In this dissertation, we first develop *Egida*, an extensible toolkit that facilitates composing implementations of rollback-recovery protocols from a library of building block modules; we then use Egida along with a few applications to study the performance protocols in practice. Before we summarize our research contributions in Section 1.2, we first provide a brief overview of existing fault-tolerance techniques in Section 1.1. Finally, Section 1.3 provides an outline of the remainder of this dissertation.

1.1 Fault-tolerance Techniques: A Perspective

Tolerating the crash of a process requires some form of replication in either space or time:

- Techniques based on replication in space mask crashes (1) by replicating the process such that at least one of the replicas is available across failures, and (2) by co-ordinating the execution of these replicas. An example of this technique is active-replication [45, 70, 72, 73].

- Techniques based on replication in time tolerate crashes by recovering the lost execution of a failed process. During failure-free execution, a process logs information about certain events it executes. This logged information is used during recovery to replay and reproduce lost execution. Examples of these techniques are primary-backup [11, 14] and rollback recovery protocols such as message-logging [6, 12, 20, 25, 37, 63, 76, 78, 79, 80, 81] and checkpointing [42, 50, 52, 60, 61, 74, 83].

These two classes of techniques trade-off the cost of providing fault-tolerance with the availability during failures. Techniques based on replication in space provide continuous availability, but (i) impose a high overhead on application performance for co-ordinating the execution of the replicas and (ii) require the use of a large number of additional resources. On the other hand, techniques based on replication in time impose low-overhead on failure-free executions and use relatively few additional resources, but reduce availability because of slow recovery. Choosing between replication in space or time for providing fault-tolerance depends on the requirements of the application. For mission-critical applications (e.g., software for monitoring the orbit of a satellite) in which a failure can lead to catastrophic consequences, tolerating arbitrary failures and providing continuous availability are crucial requirements, and replication in space is appropriate. Most of the emerging applications however, are not likely to be mission-critical. For such applications, minimizing the cost of tolerating failures will be a primary concern. Therefore, techniques based on replication in time will be appropriate.

For non mission-critical distributed applications developed using a client-server model, low-overhead fault-tolerance can be provided using a primary-backup approach. However, to exploit the computing potential of the underlying cluster-based infrastructure, most of the emerging applications are likely to go beyond the client-server model to one structured using groups of processes communicating in a peer-to-peer fashion. For such a structure, the low-overhead equivalent of primary-backup is rollback recovery protocols such as checkpointing and message-logging. In what follows, we elaborate on these protocols.

Checkpointing Checkpointing protocols, during failure-free execution, periodically save the state of each process to stable storage in what is known as a *checkpoint*. Whenever a process fails, a new process—referred to as the *recovering* process—is created, its state is restored using the latest checkpoint of its crashed counterpart, and its execution is resumed from the restored state. Though this procedure is sufficient for uni-process applications, there is a problem in the case of multi-process distributed applications: After the recovering process resumes execution, the *global system state* may not be *consistent* (The system model is formally defined in Section 2.1). A global system state includes exactly one state from each application process and it is consistent if and only if for each message delivered by a process, there exists a corresponding sender [18]. The latest consistent global state defined by a set checkpoints, exactly one from each process, is known as the *recovery line*. To restore the system to a consistent global state, it may be necessary to rollback the execution of correct processes whose state is inconsistent with the recoverable state of the failed process. Such correct processes are called *orphans*. The effect of rolling back the execution of an orphan may cause other operational processes to become orphans; rolling back the execution of these orphans may lead to further rollbacks causing the application

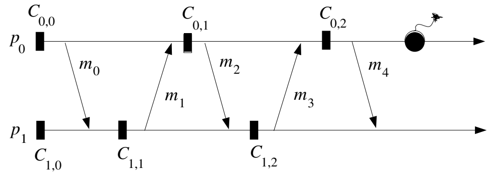


Figure 1.1: Example illustrating the domino-effect: To restore the system to a consistent state after p_0's failure, it is necessary to rollback both processes to their respective initial states.

to be rolled back to its initial state. This uncontrolled cascading of rollbacks is known as the *domino-effect* [78] and it can occur whenever processes take checkpoints independent of each other. We illustrate the domino-effect using the following example.

Example Figure 1.1 shows the execution of a system consisting of processes p_0 and p_1 that take checkpoints independent of each other. In the figure, the label $C_{i,j}$ corresponds to the j^{th} checkpoint of process $p_i (i \in \{0, 1\})$. $C_{i,j}$ also denotes the state of process p_i recorded in that checkpoint. Suppose that process p_0 fails as shown. During recovery, p_0's state is first restored to $C_{0,2}$ (i.e., its latest checkpoint). The state of p_1 becomes inconsistent with the recovered state of p_0 because p_1's state includes the delivery of message m_4 which has not been sent in state $C_{0,2}$—the recoverable state of p_0. Hence, p_1 rolls back to state $C_{1,2}$. However, the system state is still inconsistent—the scenario is same as before with the roles of p_0, p_1 inter-changed and message m_3 instead of m_4. Rolling back the execution of p_0 to $C_{0,1}$ still causes this pattern to continue. Consequently, after a sequence of cascading rollbacks, both p_0 and p_1 have to be rolled back to their initial states defined by $C_{0,0}$ and $C_{1,0}$ respectively. ■

To avoid the domino-effect, it is necessary to co-ordinate the checkpointing activities of processes. A straightforward approach is to use the Chandy-Lamport snapshot protocol [18] in which one process acts as a co-ordinator and appropriately directs all processes to take checkpoints [24, 43, 55]. Each co-ordinated checkpoint constitutes a consistent global state. Hence, even if orphans are created because of a failure, the extent of rollbacks is limited to the latest co-ordinated checkpoint.

Co-ordinated checkpointing protocols require synchronization amongst processes for creating a consistent global state and this synchronization may affect the scalability of the protocol as the number of application processes increase [30]. An alternate approach to co-ordinated checkpointing that is believed to provide better scalability and is also domino-free is communication-induced checkpointing (CIC). CIC protocols [9, 13, 30, 85] allow processes to checkpoint independently; in addition, certain checkpoint-communication patterns (e.g., $\mathcal{Z}$-cycles [56]) may induce a process to take extra checkpoints. These induced checkpoints ensure that the recovery line advances with process execution and thereby limit the extent of rollbacks.

In checkpointing protocols, to restore the system to a consistent global state after a failure, it is necessary to rollback the execution of orphans. Though the execution of the processes participating in a computation can be rolled back, the execution of the entities of the *outside world* (for instance, file-server and printer) with which a process interacts, in general cannot be rolled back. For example, if a process sends an output to a printer and fails, it may not be possible to rollback the printer to reach a consistent global state. Hence, to prevent inconsistencies, rollback-recovery protocols require a process to execute an *output-commit* protocol which ensures that the application's state in which output is generated is recoverable. In checkpointing protocols, an output-commit is typically accomplished through a co-ordinated checkpoint. The release of the output to the environment is delayed until the checkpoints have been saved to stable storage.

Message-logging To speed-up output-commits, checkpointing protocols can be used in conjunction with message-logging. Message-logging protocols are based on the observation that the lost execution of a failed process can be deterministically reproduced by replaying to the recovering process, in the same order, the results of non-deterministic events executed by its crashed counterpart prior to failure. Therefore, for an output-commit, it suffices to log to stable storage *determinants*, which are tuples that record the relative order as well as an encoding of the results of non-deterministic events a process executes.

Similar to checkpointing protocols, message-logging protocols also guarantee that when the recovery of a crashed process is complete, there are no orphans. Protocols however differ in the way they ensure this guarantee:

- Pessimistic protocols [12, 39] require that a process, prior to interacting with other processes, synchronously log on stable storage all determinants it has created so far. Thus, pessimistic protocols never create orphans.

- Optimistic protocols [20, 41, 76, 78] 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 determinant is lost when a process crashes, then orphans may be created. To reach a consistent global state, orphans must be identified and rolled back.

- Causal protocols [6, 25] combine 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. 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 further piggybacked. This guarantees that if the state of an operational process p causally depends [44] on a non-deterministic event e, then either p has a copy of e's determinant $(\#e)$ in its volatile memory or $\#e$ is logged to stable storage. This property is sufficient to restore a crashed process to a state consistent with the state of all operational processes.

A determinant in general may not contain the result of the corresponding non-deterministic event. It can be shown that if all determinants are available during recovery, then the result of any non-deterministic event can be re-generated by first rolling back the appropriate processes and then rolling them forward using the determinants. However, to speedup recovery in practice, protocols may require processes to log additional information about non-deterministic events. For instance, message-logging protocols typically log contents of messages delivered by each process: in *sender-based* protocols, messages are logged in the volatile memory of their corresponding sender; in *receiver-based* protocols messages are logged to stable storage by the receiver.

Logging the result of a non-deterministic event typically involves data copying and this introduces performance overheads. It has been shown that sender-based logging minimizes the overhead incurred on failure-free executions [26]. However, receiver-based logging is believed to simplify crash recovery. Since past research in message-logging has focussed on providing fast failure-free execution while giving little attention to recovery, sender-based logging is preferred in pessimistic [39, 79] and causal [6, 25] protocols. Though optimistic protocols can also be sender-based, to mitigate the effect of orphans on application performance, these protocols are typically receiver-based [20, 78]. Finally, pessimistic protocols based on receiver-based logging have been developed to provide fast recovery [12, 63].

1.2 Summary of Contributions

The focus of our work is to address the fault-tolerance needs of emerging non-critical distributed applications. We build up on existing rollback recovery protocols with two objectives: (1) to develop an environment that facilitates rapid implementation of any rollback recovery protocol and (2) to identify and address performance limitations of protocols when they are used in practice. To achieve these goals, we design *Egida*, an extensible toolkit for composing rollback recovery protocols; we then use Egida to provide a performance study.

To design Egida, we develop a single unifying framework for characterizing rollback recovery protocols that allows us to identify a set of functionalities that are at the core of all protocols. This framework is based on our insight that rollback recovery protocols, in spite of their diversity, share an underlying event-driven structure and are interested in the same set of "relevant" events. Protocols are therefore event-driven programs that respond in different ways whenever a relevant event occurs. We have identified five types of events that are relevant to all rollback recovery protocols: (1) non-deterministic events, (2) dependency generating events, (3) output-commit events, (4) checkpoint events, and (5) failure detection events. Informally, these events correspond to the co-ordinate axes of a five dimensional space and protocols correspond to points in this space. By simply changing the value along a single axis, it possible to move from one point in this space to another; i.e., from one protocol to another. To express rollback recovery protocols in terms of these relevant events, we have developed a simple specification language. From a specification, Egida can synthesize the corresponding protocol implementation by gluing together appropriate objects from an available library of building blocks (or core functionality modules). Consequently, Egida facilitates rapid implementation of rollback recovery protocols with minimal programming effort. An interesting side-effect of our approach is that it is now possible to develop new protocols by simply combining the building blocks in novel ways.

We have integrated Egida as a middleware with the MPICH [29] implementation of the MPI (Message Passing Interface) standard 1.1. This enables existing MPI applications to take advantage of Egida without any modifications. Conversely, the performance of rollback-recovery protocols can now be evaluated using a large set of demanding applications. We demonstrate Egida's versatility as a testbed by focussing on three fundamental performance issues: (1) recovery performance of message-logging protocols, (2) scalability of CIC protocols, and (3) the overhead imposed by message-logging protocols when processes interact with the outside world.

- *Crash Recovery:* As distributed computing becomes commonplace, there is a fresh need for techniques that impose low-overhead on failure-free executions combined with fast crash recovery. Previous studies have shown that the receiver-based pessimistic protocol imposes the highest overhead on failure-free execution, while the remaining sender-based pessimisitic, optimistic, and causal protocols impose relatively low-overhead [26]. The performance of the same protocols during recovery has been debated mostly qualitatively. We provide the first experimental evaluation of the performance of message-logging protocols during crash recovery. Our results show that, contrary to intuition, sender-based pessimistic and causal protocols outperform optimistic protocols only when a single failure is to be tolerated. For the case of multiple concurrent failures, optimistic protocols, in spite of rolling back orphan processes, can outperform sender-based pessimistic and causal protocols that do not allow fast roll-forward.

These results suggest that existing protocols leave application developers with a complex trade-off in choosing a message-logging protocol for fault-tolerance: Protocols do not simultaneously provide low-overhead on failure-free execution, fast recovery, and fault-containment. To address this complex trade-off, we develop a new class of *hybrid* protocols that for the first time combine causal logging with asynchronous receiver-based logging and provide these three desirable properties.

- *Scalability:* The merits of CIC protocols that have been argued mostly qualitatively are: (1) autonomy of allowing processes to schedule local checkpoints independently and (2) scalability with increase in the number of application processes. Our results show that CIC protocols (i) induce processes to take at least twice as many forced checkpoints when compared to the number of autonomous checkpoints and (ii) the number of forced checkpoints increases linearly with the number of processes. These suggest that the supposed merits of CIC protocols do not seem to materialize in practice and that new work may be needed to develop scalable rollback recovery protocols.

- *Interaction with outside world:* To study the effect of output-commits, we consider one component of the external environment, namely, the file system. We focus on the fault-tolerance overhead incurred by applications in which processes communicate by exchanging messages as well as reading and writing shared files. Our evaluation shows that the resulting overhead is significant and it is likely to increase as the scale of the applications and the disparity between processor and disk speeds continue to increase. We then develop a novel file-sharing protocol that virtually eliminates the incurred overhead. The protocol allows a recovering process to reproduce the interactions with the file server. In particular, it allows data lost due to a failure can be re-created during recovery. As a result, our solution introduces a novel concept—implementation in volatile memory of stable storage for files. Traditionally, a file F modified by an agent p can be shared by another agent q only after p has synchronously written F to disks at the file server [7, 33]. In our solution, no synchronous write is needed, and p can send F to q without delays. As a consequence, our solution reduces the fault-tolerance overhead for applications that communicate through both message passing and file sharing to that incurred by conventional low-overhead fault-tolerance protocols for purely message passing applications.

By building Egida and providing a performance study, our work takes some substantial steps in making rollback recovery protocols usable in practice. First, since Egida is implemented as a middleware for providing transparent fault-tolerance, it allows developers to focus solely on building their application. Egida can be used to make the application reliable, seamlessly. Second, the results of our performance studies can be used to guide developers in selecting an appropriate rollback recovery protocol to suit their particular application. Finally, Egida's event-driven architecture can be extended to handle new types of relevant events in a process' execution (e.g., thread scheduling, system calls, etc.) in natural ways.

1.3 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 describes the design and implementation of Egida. Chapter 3 describes the characteristics of applications and the experimental environment used for the performance studies. Chapter 4 contains an evaluation of the recovery performance of message-logging protocols. Chapter 5 presents an evaluation of the performance of communication-induced checkpointing protocols. Chapter 6 evaluates the failure-free performance of message-logging protocols when processes interact with the outside world. Finally, Chapter 7 presents some concluding remarks.

Chapter 2

Design of Egida

Previous research in rollback-recovery has led to the design of a diverse set of protocols (e.g., checkpointing [42, 50, 52, 60, 61, 74, 83], pessimistic [12, 39, 63, 79], optimistic [20, 38, 40, 41, 76, 78, 80, 81, 82], and causal [3, 4, 5, 6, 22, 25]). This diversity is also an opportunity for application developers to adopt a protocol that closely matches the specific needs of their application. For instance, applications that don't interact frequently with the external environment and can tolerate rollback of correct processes may use checkpointing or optimistic protocols. Pessimistic and causal protocols are better suited for applications in which communication with the environment is frequent and rollbacks are unacceptable; pessimistic protocols are preferred when fast and simple recovery is required. Applications that instead demand minimal overhead during failure-free executions would use causal or optimistic protocols.

This opportunity, however, is only available in theory and it has not been realized in practice. This is because existing systems either support only a single protocol [16, 25, 34, 48, 60, 77] or provide only the basic send and receive interface underneath which any protocol can be implemented [57, 64]. Hence, these systems leave developers and users with three choices in selecting the protocol to use with their application: (1) implement the appropriate protocol by themselves, (2) use a protocol that may not be the best fit for their particular application, or (3) renounce fault-tolerance. To take advantage of the flexibility provided by rollback-recovery protocols, a fault-tolerance toolkit that simplifies protocol implementation is desirable.

Toolkit-based approaches for implementing fault-tolerance protocols have been pursued in the past. For instance, ISIS [10], Horus [66, 67], and Transis [21] simplify the implementation of fault-tolerance protocols based on active-replication [73]. The toolkit we want to build is instead based on rollback recovery. The challenge in building our toolkit is that, because of the diversity of rollback-recovery protocols, it is necessary to develop a unified framework that would allow us to emphasize the similarities that do exist among these protocols. We address this challenge by identifying fundamental concepts that are at the core of all rollback recovery protocols. We build our toolkit, *Egida*, by instantiating the core concepts in fine-granularity re-usable modules that facilitate rapid protocol implementation.

The remainder of this chapter is organized as follows. We first present in Section 2.1 the system model that underlies Egida's design. In that section, we also introduce terminology that used in this dissertation. In Section 2.2, we identify a set functionalities that are at the core of all log-based rollback recovery protocols and present a grammar for synthesizing protocols from the core components. Section 2.3 describes Egida's architecture. Section 2.4 discusses related work. Finally, Section 2.5 summarizes our contributions.

2.1 System Model

We assume an asynchronous distributed system consisting of a set of n processes. Processes communicate using both message passing and file sharing. Messages are exchanged over channels that can fail by transiently losing some messages. We assume that each message sent by a process is eventually received by their corresponding receiver. Files are shared according to an ownership-based consistency protocol [7, 33]. Specifically, the file server supports shared read-ownership, and exclusive write-ownership. At any point in time, the content of a file is uniquely identified by its *version*. We denote version v of file F by $F.v$. Prior to F's creation, its version is denoted by $F.\bot$. The latest version of F is denoted by $F.\top$. To read F, a process sends to the file server a read request specifying the version v. On receiving that request, the file server checks whether $F.v$ exists and if $v \neq \bot$, the file server returns the content of $F.v$; otherwise, the file server returns an error. For writing to F, a process sends to the file server a message containing the data to be written. On receiving that message, the file server uses the content of that message to modify $F.\top$ and thereby create a new version of F. This new version is now $F.\top$.

The execution of the system is represented by a *run*, which is an irreflexive partial ordering of send, receive, read, write, and local events, ordered by potential causality [44]. Let e_1, e_2 correspond to two events in an execution. We denote $e_1 \prec e_2$ (read "e_1 immediately precedes e_2") if one of the following two conditions hold:

1. e_1, e_2 are executed by the same process and e_1 precedes e_2.

2. e_1 is a message send event and e_2 is the corresponding receive event.

The transitive closure of the relation $\prec$ is the *happened-before* [44] relation denoted by $\rightarrow$. If $e_1 \rightarrow e_2$, then we say that "e_1 happens before e_2", "e_1 causally precedes e_2", or "e_2 causally depends on e_1".

For each process p, a special class of events local to p are called *deliver events*. These events correspond to the delivery of a message to the application that p is part of. For any message m sent from process p_1 to p_2, we assume that p_2 delivers m at most once. Furthermore, we assume that a correct process will eventually deliver all messages it has received.

In a run, process execution consists of deterministic and non-deterministic events. A non-deterministic event is an event whose outcome may change for different executions of the same program. In our case, deliver, read and write events are non-deterministic because the order in which a process receives messages and the file versions it accesses are execution-dependent. Send events and other local events are instead deterministic. Process execution is *piecewise deterministic* [79]: It consists of a sequence of deterministic intervals of execution, joined by non-deterministic events. At any point during the execution, the *state* of a process is a mapping of program variables and implicit variables (such as program counters) to their current values. We assume that the state of the process does not include the state of the underlying communication system, such as the queue of messages that have been received but not yet delivered to the process. Given the initial state of each process and the non-deterministic events that start each of deterministic intervals, the remaining states in their execution are uniquely determined.

Given the states s_p and s_q of two processes p and q, $p \neq q$, we define the following notions of consistency for s_p and s_q (or, simply, for p and q):

- p and q are *mutually message-consistent* if all messages from q that p has delivered during its execution up to s_p were sent by q during its execution up to s_q, and vice versa.

- p and q are *mutually file-consistent*, for all versions v and files F, if p has read file $F.v$ written by q during its execution up to s_p, then q has written $F.v$ during its execution up to s_q, and vice versa.

Two processes p and q are *mutually consistent* if they are both mutually message- and file-consistent. A collection of states, one from each process, is a *consistent global state* if all pairs of states are mutually consistent [18]; otherwise it is *inconsistent*.

The $\prec$ relation defined over events can be naturally extended to include process states. In a run, the execution of process p consists of a sequence of states beginning from its initial state. When p executes event e in state s_e, e causes p's state to transition from s_e to $s_{e'}$. Now,

- for all events e, $s_e \prec s_{e'}$

- for all pairs of events e_1, e_2, if $e_1 \prec e_2$ then $s_{e_1} \prec s_{e_2}$.

The transitive closure of $\prec$ over states is the *happened-before* relation $\rightarrow$ defined over states.

We assume that processes fail according to the fail-stop model [71]. In this model, processes fail only by halting, and a process crash is eventually detected by all non-faulty processes. We assume that a failed process is eventually re-started and it resumes computation. We also assume that *stable storage* [28] is available throughout the system and it persists across failures.

2.2 Deconstructing Log-Based Rollback-Recovery Protocols

As a first step towards identifying the core ideas on which to build our toolkit, we focus on the consistency condition that is common to all rollback-recovery protocols. This condition, known as the *no-orphans consistency condition* [2], guarantees that when a failed process completes recovery, there are no orphans. To implement this condition, whenever a process fails, it is simply re-started and restored to the initial state of its crashed counterpart; and then the execution is rolled forward. If the process' execution prior to failure was completely deterministic, then its execution during recovery will reproduce the computation until the point of crash. In this scenario, the no orphans consistency condition trivially holds.

The simple procedure we just described for recovering lost execution may not always be sufficient. This is because of two reasons:

1. Processes may execute non-deterministic events prior to a crash.

2. Processes may communicate with other processes or with the external environment. As a result, the states of the other processes or of the external environment causally depend on non-deterministic events executed by a process. For example, a process may read the value of the system's clock and send a message; when that message is subsequently delivered, the state of the receiver becomes causally dependent on the value returned by the system's clock.

Consequently, orphans may be created when a failed process completes recovery: if the results of non-deterministic events executed by a process before crashing are not reproduced during recovery, then causal dependencies that existed may not be re-created during recovery. Hence, to enforce the no orphans consistency condition, it is crucial that all rollback recovery protocols handle non-determinism in a process' execution. Rollback-recovery protocols use two different approaches to do so:

- Message-logging protocols identify the specific non-deterministic events executed by each process and log their correspondng result during failure-free execution. The logged information is used during recovery to replay and to reproduce lost execution. However, if this information is lost in a crash, results of non-deterministic events executed before the crash may not be reproduced during recovery. Consequently, in some message-logging protocols [20, 38, 40, 41, 76, 78, 80, 81, 82], orphan processes may be created; these processes are identified and their execution is rolled back.

- Checkpointing protocols do not explicitly identify the non-deterministic events executed by processes. Instead, these protocols treat any event executed by a process as potentially non-deterministic and therefore assume that its corresponding result, in general, will not be reproducible during recovery. Hence, unless the recovering process is restored to the state of its crashed counterpart in which the failure occurred, causal dependencies will not be re-created. Consequently, operational processes may become orphans; to restore the system to a consistent state, these processes are identified and their execution is rolled back to the recovery line.

In either approach, rollback-recovery protocols handle non-determinism by taking appropriate actions in response to events in an execution. Protocols can therefore be characterized as event-driven programs that respond in different ways when some events occur. Hence, we design Egida using an event-driven architecture. Our key insight in designing Egida is that all protocols are actually interested in the same set of "relevant" events. In this section we identify the relevant events and present a language that can be used to specify a protocol in terms of which actions it takes in response to each of these events.

2.2.1 Log-based Rollback-Recovery Protocols as Event-Driven Programs

There are five types of events that are relevant to all rollback-recovery protocols: (1) non-deterministic events, (2) dependency-generating events, (3) output-commit events, (4) checkpointing events, and (5) failure-detection events.

1. **Non-deterministic events**: In response to a non-deterministic event executed by a process, log-based protocols use determinants to record the information necessary to reproduce the event's result during recovery. If determinants are lost because of a failure, correct processes whose state depends on the unrecoverable non-deterministic events must be rolled back. Restoring the system to a consistent state while limiting the extent of rollbacks depends on the ability of the rollback-recovery protocol to replay the determinants of the non-deterministic events executed by the failed processes before crashing. Hence, determinants must be saved to stable storage so that they are available during recovery. In practice, different protocols choose different ways to log determinants and rely on different implementations of stable storage.

   An example of a non-deterministic event is a message deliver event. This is because its outcome depends on many factors, including process scheduling, routing, and flow control. Thus, a recovering process may not produce the same run upon recovery even if the same set of messages are sent to it.

2. **Dependency-generating events**: These events can increase the number of processes that depend on the non-deterministic events executed by a process. In response to a dependency-generating event, protocols track causal dependencies that may be generated. To do so, protocols typically append to application messages dependency-tracking information—logging protocols piggyback logical clocks [44] or vector clocks [53],

and checkpointing protocols piggyback an index that records the number of checkpoints taken by a process. In addition, to speedup the replay of non-deterministic events, it may be desirable to log some information on executing a dependency-generating event. For instance, in sender-based message-logging protocols processes log the content of each message they send.

An example of a dependency-generating event in message-passing applications is a message-send event. The send event can create dependencies between non-deterministic events executed by the sender and the state of the receiver.

3. **Output-commit events**: These events can make the external environment depend on the non-deterministic events executed by a process.

   Since it is not reasonable to expect the external environment to roll back in response to a process crash, log-based rollback recovery protocols invoke *output commit* procedures that write synchronously to stable storage the determinants of all non-deterministic events that precede the communication with the external environment, thus making them recoverable. Depending on how determinants are logged, this procedure may require co-ordination among processes.

   An example of an output-commit event executed by a process is a file write event. The write event can create dependencies between non-deterministic events executed by the process and the state of the file system.

4. **Checkpointing events**: These events instruct the protocols to write to stable storage the state of one or more processes. These events can either be transparent to the application (e.g. a signal from a timer indicating that some pre-determined time has elapsed since the last checkpoint) or be explicitly driven by the application.

   Checkpoints can be either independent or co-ordinated. Processes may checkpoint their state incrementally, and checkpoints can be saved to stable storage either synchronously or asynchronously.

5. **Failure-detection events**: These events are generated on detecting the failure of one or more processes.

   In response to these events, faulty processes must be restarted, restored to a previously checkpointed state and rolled forward. In turn, correct orphan processes may need to be detected and rolled back. Different protocols implement different strategies to collect the determinants to be replayed during recovery and to detect and roll back orphans.

Implementing a specific protocol therefore amounts to selecting the set of actions performed in response to each relevant event. We now show how a simple language can be used to specify these choices. In Section 2.3 we show how protocols can be synthesized starting from these specifications.

2.2.2 Specifying Design Choices in Rollback-Recovery Protocols

Figure 2.1 shows a simple language that we use to specify rollback-recovery protocols. The language reflects the discussion of Section 2.2.1. A protocol is defined in terms of the actions it takes in response to non-deterministic events, dependency generating events, output commit events, checkpointing events and failure-detection events. To define a protocol completely, it is necessary to instantiate a set of variables which specify, for instance, the set of non-deterministic events, the form of their determinant, the implementation of stable storage, etc. Figure 2.2 shows a set of representative instantiations of these variables used in several existing protocols.

To facilitate rapid specification of rollback-recovery protocols, we enhance the language to allow for default values for each variable. Now, a specification has the effect of modifying only the specified values for variables from their default values. Figure 2.3 shows the specification corresponding to the "default protocol" which is an independent checkpointing protocol. The specification shows that (1) no information is logged on a receive, (2) a process piggybacks its vector clock on out-going messages, (3) whenever a process writes to a file, the write triggers an output-commit which is a co-ordinated checkpoint, (4) incremental checkpoints are saved synchronously on stable storage, (5) the checkpoint counter is incremented after a checkpoint is saved, (6) to compute the recovery line, processes exchange their vector clocks corresponding to the state saved in each checkpoint, (7) no actions are executed on a rollback, (8) for orphan detection, a recovering process broadcasts the logical clock corresponding to its recoverable state, and (9) no actions are executed at the end of roll-forward phase of recovery.

Our language can be used for more than just specifying existing recovery protocols. Once deconstructing rollback-recovery protocols becomes possible, it is then easy to reassemble the fundamental components in novel ways to implement *new* protocols. An example is given in Figure 2.4(b), which shows a specification of a sender-based pessimistic message-logging protocol that tolerates f concurrent failures and implements stable storage by replicating data in the volatile memory of processes. In this protocol, which to our knowledge has not appeared in the literature, a process executing a non-deterministic event broadcasts the corresponding determinant to the other processes and does not send any application message until it has received at least f acknowledgments to its broadcast. This protocol is a variant of Strom, Bacon, and Yemini's sender-based pessimistic protocol [79], whose specification is shown in Figure 2.4(a).

Our language, while allowing specification of rollback-recovery protocols in terms of the relevant events, does not guarantee that a specification is indeed correct. Tools for verifying the correctness of a specification are beyond the scope of this dissertation.

Protocol	:=	⟨non-det-event-stmt⟩ ⟨dep-gen-event-stmt⟩ ⟨output-commit-event-stmt⟩ ⟨ckpt-stmt⟩ ⟨recovery-stmt⟩
⟨non-det-event-stmt⟩	:=	[⟨event⟩ : *determinant* : ⟨determinant-structure⟩ ⟨*Log* ⟨event-info⟩ *on* {⟨volatile-storage⟩ | ⟨stable-storage⟩ | ⟨volatile-storage⟩ , ⟨stable-storage⟩ *Flush* : ⟨timer-sched-policy⟩} ⟩*]*
⟨dep-gen-event-stmt⟩	:=	[⟨event⟩ : {⟨piggyback-stmt⟩ | ⟨logging-stmt⟩ | ⟨piggyback-stmt⟩ ⟨logging-stmt⟩}]*
⟨logging-stmt⟩	:=	*Log* ⟨event-info-list⟩ *on* ⟨volatile-storage⟩
⟨event-info-list⟩	:=	⟨event-info⟩ | ⟨event-info⟩, ⟨event-info-list⟩
⟨piggyback-stmt⟩	:=	*Piggyback* : {⟨pb-log-value-list⟩ | ⟨pb-ckpt-value-list⟩ | ⟨pb-log-value-list⟩, ⟨pb-ckpt-value-list⟩}
⟨pb-log-value-list⟩	:=	⟨pb-log-value⟩ | ⟨pb-log-value⟩, ⟨pb-log-value-list⟩
⟨pb-ckpt-value-list⟩	:=	⟨pb-ckpt-value⟩ | ⟨pb-ckpt-value⟩, ⟨pb-ckpt-value-list⟩
⟨output-commit-event-stmt⟩	:=	[⟨event⟩ : *Output commit* : ⟨what-to-output-commit⟩ *Protocol* : ⟨output-commit-proto⟩]*
⟨ckpt-stmt⟩	:=	[*Checkpoint* : ⟨ckpt-proto⟩, ⟨how-to-ckpt⟩ *on* ⟨stable-storage⟩ *Implementation* : ⟨ckpt-impl⟩ *Scheduling policy* : ⟨timer-sched-policy⟩ *On checkpoint* : { *Update* ⟨pb-ckpt-value-list⟩ | *Skip*} ⟨*Protocol to compute recovery line* : ⟨compute-recovery-line-proto⟩⟩$_{opt}$]$_{opt}$
⟨recovery-stmt⟩	:=	[*Recovery* : {⟨recovery-get-det-stmt⟩ ⟨orphan-detection-stmt⟩ ⟨rollback-stmt⟩ ⟨rollfwd-done-stmt⟩}$^+$]$_{opt}$
⟨recovery-get-det-stmt⟩	:=	*Protocol to retrieve determinants* : ⟨recovery-get-det-proto⟩
⟨orphan-detection-stmt⟩	:=	*Protocol to detect orphans* : ⟨orphan-detection-proto⟩
⟨rollback-stmt⟩	:=	*On rollback* : {*Flush* ⟨event-info-list⟩ | *Skip*}
⟨rollfwd-done-stmt⟩	:=	*Roll-forward done* : {*Flush* ⟨event-info-list⟩ | *Skip*}

Figure 2.1: A grammar for specifying rollback-recovery protocols. In the grammar, we use the following notational convention: ⟨x⟩ means that x is a production rule, ⟨y⟩ means that y is a variable that has to be mapped to a specific value, ⟨z⟩$_{opt}$ means that ⟨z⟩ is an optional statement, and keywords are in *italics*.

⟨event⟩	:=	send $\mid$ receive $\mid$ read $\mid$ write
⟨determinant-structure⟩	:=	{source, sesn, dest, desn}
⟨output-commit-proto⟩	:=	independent $\mid$ co-ordinated
⟨what-to-output-commit⟩	:=	determinants $\mid$ checkpoint
⟨event-info⟩	:=	determinant $\mid$ message
⟨volatile-storage⟩	:=	local disk $\mid$ volatile memory of self
⟨stable-storage⟩	:=	local disk $\mid$ NFS disk $\mid$ volatile memory of processes
⟨pb-log-value⟩	:=	determinants⟨, ⟨dependency matrix $\mid$ stability matrix $\mid$ stability vector⟩⟩$_{opt}$ $\mid$ vector clock $\mid$ logical clock
⟨pb-ckpt-value⟩	:=	checkpoint counter $\mid${checkpoint vector, pred matrix} $\mid$ {checkpoint counter, checkpoint vector, taken, greater}
⟨ckpt-proto⟩	:=	independent $\mid$ co-ordinated
⟨how-to-ckpt⟩	:=	synchronous $\mid$ asynchronous
⟨ckpt-impl⟩	:=	full $\mid$ incremental
⟨timer-sched-policy⟩	:=	periodic $\mid$ random
⟨recovery-get-det-proto⟩	:=	centralized $\mid$ distributed
⟨orphan-detection-proto⟩	:=	broadcast logical clock $\mid$ exchange vector clock
⟨compute-recovery-line-proto⟩	:=	checkpoint graph $\mid$ exchange vector clock

Figure 2.2: Representative instantiations for the variables defined in the grammar shown in Figure 2.1. For each variable we also provide a NULL instantiation which, for the sake of clarity, is not shown in this figure.

/* `comment:` ⟨non-det-event-stmt⟩ */
receive:
  determinant : NULL
  Log determinant on NULL
  Log message on NULL

read:
  determinant : NULL
  Log determinant on NULL
  Log data on NULL

/*`comment:` ⟨dep-gen-event-stmt⟩} */
send:
  Piggyback : vector clock
  Log message on NULL

/* `comment:` ⟨output-commit-stmt⟩} */
write:
  Output commit : checkpoint
  Protocol : co-ordinated
/* `comment:` ⟨ckpt-stmt⟩} */
Checkpoint : independent, synchronous on NFS disk
Implementation : incremental
Scheduling policy : periodic
On checkpoint : Skip
Protocol to compute recovery line : exchange vector clock

/* `comment:` ⟨recovery-stmt⟩} */
Recovery :
  Protocol to retrieve determinants : NULL
  Protocol to detect orphans : broadcast logical clock
  On rollback : Skip
  Roll-forward done : Skip

Figure 2.3: Specification for the “default protocol”.

/* comment: ⟨non-det-event-stmt⟩ */
receive:
  determinant : {source, sesn, dest, desn}
  Log determinant on
    NFS disk

/*comment: ⟨dep-gen-event-stmt⟩} */
send:
  Piggyback : NULL
  Log message on
    volatile memory of self

/* comment: ⟨ckpt-stmt⟩} */
Checkpoint : independent, asynchronous
  on NFS disk

/* comment: ⟨recovery-stmt⟩} */
Recovery :
  Protocol to retrieve determinants : distributed
  Protocol to detect orphans : NULL

(a)

/* comment: ⟨non-det-event-stmt⟩ */
receive:
  determinant : {source, sesn, dest, desn}
  Log determinant on
    volatile memory of processes

/* comment: ⟨dep-gen-event-stmt⟩ */
send:
  Piggyback : NULL
  Log message on
    volatile memory of self

/* comment: ⟨ckpt-stmt⟩ */
Checkpoint : independent, asynchronous
  on NFS disk

/* comment: ⟨recovery-stmt⟩} */
Recovery :
  Protocol to retrieve determinants : distributed
  Protocol to detect orphans : NULL

(b)

Figure 2.4: Specification of (a) Strom, Bacon, and Yemini's sender-based pessimistic message logging protocol and (b) a novel sender-based pessimistic protocol. In both of these specifications we only show the changes to the default protocol from Figure 2.3.

2.3 The Architecture of Egida

Figure 2.1 defines the structure of log-based rollback recovery protocols, while the variables in Figure 2.2 identify the "building blocks" which, when incorporated into the protocol structure, yield different rollback recovery protocols. Egida instantiates these building blocks in a modular, object-oriented architecture. We begin this section by providing an overview of Egida's architecture and then describe the functionality of each building block module and the methods it exports. We then explain how these modules are invoked in response to events and how they can be composed to synthesize rollback-recovery protocols.

2.3.1 Overview

We design Egida's architecture in two steps. First, we focus on how the relevant events are generated and tracked. Second, we build modules for executing actions in response to relevant events.

Three of the five types of relevant events—i.e., non-deterministic, dependency-generating, and output-commit events—are generated by application processes in their execution. Checkpoint events are generated in different ways: by an application process, e.g., whenever the process determines that saving a checkpoint can incur minimal overhead; by the expiration of a timer; and, by rollback-recovery protocols in response to specific patterns of communication, e.g., in communication-induced checkpointing protocols [9, 13, 30, 85] for preventing the occurrence of a $\mathcal{Z}$-cycle [56]. Failure-detection events are generated upon process crashes by some entity that detects failures.

To execute the appropriate response to a relevant event, we use event handlers. An event handler is defined using the EventHandler module. The mechanism used to invoke methods in the appropriate EventHandler module depends on the specific event being handled. For events generated by application processes, we need (1) to intercept the application-level call responsible for generating the event and (2) to map the call to the event's handler. Both of these actions are performed by the Application Programming

Interface (API) module. The Timer module generates timer interrupts for performing periodic actions such as checkpointing and flushing of volatile logs to stable storage. The FailureDetector module detects process failures, re-starts crashed processes, and generates failure-detection events. To allow processes to communicate by exchanging messages, we define (1) the Network module that provides the functionality for sending and receiving messages and (2) the MsgDispatch module that processes incoming messages. In addition to these modules, we define the Init module that initializes Egida's data structures when a process is started.

We build the remaining components of Egida's architecture by focusing on the actions of the event handler of each relevant event:

1. **Non-deterministic events**: When a process executes a non-deterministic event, the event handler creates and logs the corresponding determinant and logs the event's result, if necessary. In both cases, the logged information is saved on storage.

   The Determinant module provides the functionality for creating determinants. The Log module encapsulates the functionality for logging determinants and event results. The storage abstraction is defined by the Storage module which supports both volatile storage and stable storage.

2. **Dependency-generating events**: When a process executes a dependency-generating event, the event handler piggybacks dependency-tracking information. We define the Piggyback module to provide the information to be piggybacked.

   In addition to tracking causal dependencies, the event handler may need to log information on executing a dependency-generating event. For instance, in message-logging protocols, a message send is a dependency-generating event and some protocols (for example, [6, 39]) require the sender to log a copy of the sent message. Logging information for dependency-generating events and non-deterministic events is similar. Hence, the functionality provided by Log and Storage modules can be re-used.

3. **Output-commit events**: For an output-commit event, the corresponding event handler executes an output-commit. An output-commit involves saving on stable storage information to make the output-commit event recoverable. The saved information consists either of determinants from volatile logs or of a checkpoint. The functionality for saving determinants and taking checkpoints are provided, respectively, by the Log module and by the HowToCkpt module which is discussed below.

   Determining *how* to execute an output-commit, i.e., either independently or in a co-ordinated manner, is performed by the HowToOutputCommit module. Determining *what* to save for an output-commit is performed by the WhatToOutputCommit module.

4. **Checkpoint events**: When a checkpoint event occurs, the event handler takes a checkpoint. A checkpoint is a snapshot of the state of a process. To take a checkpoint, the event handler determines (1) the protocol for saving the checkpoint, i.e., co-ordinated or independent, (2) the actions that must be executed on a checkpoint, e.g., scheduling the next checkpoint, (3) the policy for saving a checkpoint, i.e., synchronously or asynchronously, and (4) the process state to be saved in a checkpoint, i.e., full or incremental. The methods for executing each of these four tasks is provided, respectively, by the modules CkptProtocol, CkptActions, HowToCkpt, and Ckpt. The Ckpt module also provides methods for restoring a process state to a previous checkpoint.

   After a checkpoint is saved on stable storage, Ckpt can initiate garbage collection, which may involve discarding old checkpoints and purging volatile logs. Before discarding checkpoints, it may be necessary to compute the recovery line. The protocol for computing the recovery line is encapsulated by the RecoveryLine module. Finally, Storage can be re-used to save a checkpoint on stable storage.

   Determining the policy to schedule checkpoint events using timers is performed by the TimerSchedPolicy module.

5. **Failure-detection events**: On detecting a failure, the failure detector re-starts the crashed process and generates a failure-detection event. The corresponding event handler of the recovering process performs three actions before roll-forward: First, it restores the process state to a previous checkpoint; second, it retrieves the determinants that have to be replayed during recovery; and finally, it initiates orphan detection. The HowToCkpt module provides the method for executing the first of these three actions; for executing the remaining two actions, we define two modules, namely, CollectRecoveryInfo and OrphanDetection.

   To roll back an orphan process, we define the Rollback module. Some protocols [20, 81] require a correct process that has become an orphan to execute specific actions before rolling back. These actions are encapsulated in the RollbackActions module. After executing these actions, the Rollback module identifies the checkpoint to which the process has to be rolled back by performing a recovery-line computation; it then uses the HowToCkpt module to roll back to that checkpoint.

   During the roll-forward phase of recovery, the methods for replaying the results of non-deterministic events is defined by the RollForward module. Finally, at the end of roll-forward, protocols may require a process to execute specific actions. These actions are defined in the RollForwardDone module.

2.3.2 Modules and Exported Methods

The modules discussed in Section 2.3.1 and the methods they export are as follows.

The Init module initializes Egida's data-structures when a process is started. Also, this module generates a failure event upon the re-start of a crashed process. Init exports a single method, *Init*.

The API module exports the methods for intercepting application-level calls to all the relevant events. It exports five methods, namely, (1) *Send*, (2) *Receive*, (3) *Read*, (4) *Write*, and (5) *Checkpoint*.

The FailureDetector module implements the protocol for failure-detection and restarts crashed processes. It does not export any methods.

The Network module implements the functionality for sending and receiving messages. It exports two methods, namely, *Send* and *Receive*. To process incoming messages, *Receive* invokes the MsgDispatch module's *ProcessIncomingMsg* method.

MsgDispatch dispatches incoming messages. It exports a single method, *ProcessIncomingMsg*. Messages are handled appropriately depending on their type:

- *Application messages*: These messages are queued for delivery to the application.
- *Control messages*: These messages are dispatched to the appropriate Egidamodule. For instance, messages of type "output-commit" are dispatched to the HowToOutputCommit module.

The EventHandler module exports methods for handling the relevant events. There are separate modules for each class of events:

- **Dependency-generating events**: In our implementation, we assume that the only event that generates causal dependencies is a message send event. Hence, the EventHandler module for dependency-generating events exports a single method, *Send*. It uses the methods exported by (1) the Piggyback module to obtain the information to be piggybacked, (2) the Network module to send the message using the communication sub-system, (3) the LogEventInfo module to log a copy of the sent message.
- **Non-deterministic events**: In our implementation, we assume that there are two sources of non-determinism in a process' execution: (1) the order in which messages are delivered to the application and (2) the content of a file read by a process (i.e., when a process accesses a shared file, the version it reads is non-deterministic). Hence, the EventHandler module for non-deterministic events exports two methods, namely, *Receive* and *Read*.

  The actions executed by *Receive* and *Read* depend on the state of the process:

- During failure-free execution: *Receive* retrieves a message from the queue of incoming messages. It uses the methods exported by (1) the Piggyback module to process the piggybacked information, (2) the Determinant module to create a receive determinant, (3) LogDeterminant and LogEventInfo module's to respectively log the determinant and the message content. Finally, *Receive* delivers that message to the application process.

  Read reads data from the specified file. Similar to *Receive*, *Read* logs a read determinant and a copy of the data that is read.

- During recovery: *Receive* and *Read* obtain their corresponding result for replay from the RollForward module.

- **Output-commit events**: In our implementation, we assume that the only output-commit event is when the application writes data to a file. Hence, the EventHandler module for output-commit events exports a single method, *Write*. To execute an output-commit, *Write* uses the methods exported by the HowToOutputCommit module.

- **Checkpoint events**: The EventHandler module for checkpoint events exports a single method, *Checkpoint*.

  The *Checkpoint* method triggers a checkpoint by invoking the CkptProtocol module's *CkptStart* method.

- **Failure-detection events**: The EventHandler module for failure-detection events exports two methods: (1) *Failure* and (2) *Recover*.

  The *Failure* method is the event handler for failure-detection events. This method uses the HowToCkpt module to restore the process state to a previous checkpoint. After the state is restored, unlike in normal method calls, control *does not* return to the *Failure* method; instead, control returns to the point where the checkpoint was taken in the Ckpt module. Subsequently, from the Ckpt module, the *Recover* method is invoked to trigger the appropriate recovery mechanisms, namely, (1) retrieving logged information that is needed for replay during recovery and (2) performing orphan detection. For these two actions, *Recover* invokes, respectively, methods exported by modules CollectRecoveryInfo and OrphanDetection.

The Timer module generates a timer interrupt at a specified time interval. It exports a single method, *SetTimer*, which schedules a timer interrupt. For generating timer interrupts, our implementation uses the timer facility provided by the operating system. The Timer module contains a method (*TimerInterrupt*) that is invoked from the operating system whenever a scheduled timer expires. The Timer module supports scheduling of multiple timer interrupts and the *TimerInterrupt* method uses callbacks to notify the expiration of timers. Hence, whenever a timer interrupt is scheduled, it is necessary to provide the callback which will be appropriately invoked by the *TimerInterrupt* method.

The TimerSchedPolicy module contains the policy that determines when to schedule the next checkpoint or flushing of volatile logs to stable storage, e.g., periodically or a-periodically. It exports a single method, *SetTimer*. For setting timer interrupts, *SetTimer* invokes Timer module's *SetTimer* method.

The Determinant module creates a determinant for each non-deterministic event. It exports a single method, *CreateDeterminant*.

The Log module saves to storage the determinant and the data associated with an event. It exports five methods: (1) *Log*, (2) *Retrieve*, (3) *Flush*, (4) *GarbageCollect*, and (5) *TimerInterrupt*. There are two instantiations of the Log module: (1) LogDeterminant module that logs determinants and (2) LogEventInfo module that logs event data. In both instantiations, the *Flush* method is invoked whenever it is necessary to perform a synchronous transfer of data logged in volatile storage to stable storage (for example, when an output-commit is executed). The *TimerInterrupt* method is the callback that gets invoked from the Timer module when a scheduled timer interrupt occurs. *TimerInterrupt* first triggers a flush by invoking *Flush* and then schedules the next timer-driven flush by invoking the *SetTimer* method in module TimerSchedPolicy.

The Storage module implements the abstraction of storage for determinants and data associated with an event. This module exports four methods: (1) *Read*, (2) *Write*, (3) *Flush*, and (4) *GarbageCollect*. There are two instantiations of this module, namely, VolatileStorage and StableStorage. In both instantiations, the *Write* method writes data synchronously. In contrast, the implementation of the *Flush* method depends on the specific instantiation of Storage: In VolatileStorage, *Flush* writes logged data to StableStorage; in StableStorage, *Flush* is an empty method because the *Write* method itself synchronously writes data to stable storage,

The Piggyback module exports methods that are invoked by event handlers for obtaining and processing information piggybacked on application messages. It exports two methods, (1) *GetPiggyback*, (2) *ProcessPiggyback*. There are two instantiations of this module: PiggybackLog and PiggybackCkpt; they provide, respectively, the information piggybacked by message-logging and checkpointing protocols.

The HowToOutputCommit module determines how to save the information necessary to make an output-commit event recoverable. It exports two methods: (1) *OutputCommit* and (2) *ProcessMsg*. To execute a co-ordinated output-commit, the output-commit protocol causes control messages to be exchanged among processes. These messages are handled by the *ProcessMsg* method which gets invoked from the MsgDispatch module.

The WhatToOutputCommit module determines the information that is saved for an output-commit. It exports a single method, *OutputCommit*.

The CkptProtocol module contains the implementation of the checkpointing protocol (either independent or co-ordinated). This module exports three methods, (1) *CkptStart*, (2) *CkptDone*, and (3) *ProcessMsg*. Since checkpoints may be asynchronously saved to stable storage, we use a two-step procedure to execute actions before and after a checkpoint. First, *CkptStart* uses the CkptActions module to execute actions before triggering a checkpoint; second, after a checkpoint is saved to storage, from HowToCkpt the *CkptDone* method is invoked. This method uses the CkptActions module to execute actions after a checkpoint. *CkptStart* uses the HowToCkpt module to take a checkpoint. To execute a co-ordinated checkpoint, the checkpoint protocol causes control messages to be exchanged among processes. These messages are handled by the *ProcessMsg* method which gets invoked from the MsgDispatch module.

The CkptActions module exports the methods for executing actions before and after a checkpoint is saved on storage. It exports two methods, (1) *PreCkptActions* and (2) *PostCkptActions*. These methods are invoked from the CkptProtocol module, respectively, before and after a checkpoint is saved to storage. These methods contain actions such as updating data-structures (for instance, in communication-induced checkpointing protocols [9, 13, 30, 85]), scheduling timer-driven checkpoints using the TimerSchedPolicy module's *SetTimer* method, and garbage collection of logs and checkpoint files.

The HowToCkpt module exports the methods for checkpointing: (1) *TakeCkpt*, (2) *RestoreFromCkpt*, (3) *GarbageCollect*, and (4) *Flush*. *TakeCkpt* determines whether checkpoints are saved synchronously or asynchronously. *Flush* blocks until the checkpoint initiated by a prior call to *TakeCkpt* is saved on storage. Note that *Flush* may incur the blocking only when checkpoints are saved to storage asynchronously.

The Ckpt module contains the actual implementation of checkpointing (for full or incremental checkpointing). It exports three methods, namely, (1) *TakeCkpt*, (2) *RestoreFromCkpt*, and (3) *GarbageCollect*.

The RecoveryLine module implements the protocol for computing the recovery line. It exports a single method, *ComputeRecoveryLine*.

The CollectRecoveryInfo module collects the information needed for replay during recovery. It exports two methods: (1) *RetrieveRecoveryInfo* and (2) *ProcessMsg*. In a recovering process, the *RetrieveRecoveryInfo* method retrieves the recovery information—determinants and results—in two steps: First, it invokes the *Retrieve* method exported by

LogDeterminant and LogEventInfo modules for retrieving the information logged locally; second, it sends control messages to the remaining processes for obtaining recovery information that may be available from their logs. In each operational process, these control messages are processed by the *ProcessMsg* method which is invoked from the MsgDispatch module. To retrieve information logged for the recovering process, *ProcessMsg* invokes the *Retrieve* method exported by LogDeterminant and LogEventInfo modules.

The OrphanDetection module implements the protocol for orphan detection. It exports two methods: (1) *DetectOrphans* and (2) *ProcessMsg*. The protocol for detecting orphans may cause control messages to be exchanged among processes. These messages are handled by the *ProcessMsg* method which is invoked from the MsgDispatch module.

The Rollback module is used to roll back the execution of an operational process. It exports a single method, *Rollback*. When the OrphanDetection module determines that the process is an orphan, it invokes the *Rollback* method. The *Rollback* method performs three actions: First, it identifies the checkpoint to which the process has to be rolled back—either by using protocol-specific data-structures or by performing a recovery-line computation; second, it uses the RollbackActions module for executing the actions prior to rolling back; and finally, it uses the HowToCkpt module to perform the roll back.

The RollbackActions module contains the actions that are executed by an orphan prior to its roll back. This module exports a single method, *RollbackActions*.

The RollForward module contains methods for replaying non-deterministic events during the roll-forward phase of recovery. It exports two methods, *ReplayDeliver* and *ReplayRead*. The actions executed by both of these methods is similar. In both cases, the *Retrieve* methods exported by LogDeterminant and LogEventInfo modules are invoked, respectively, for retrieving a determinant and its corresponding result.

The RollForwardDone module contains the actions that are executed by a process at the end of the roll-forward phase of recovery. This module exports a single method, *RollForwardDone*.

2.3.3 Module Invocation

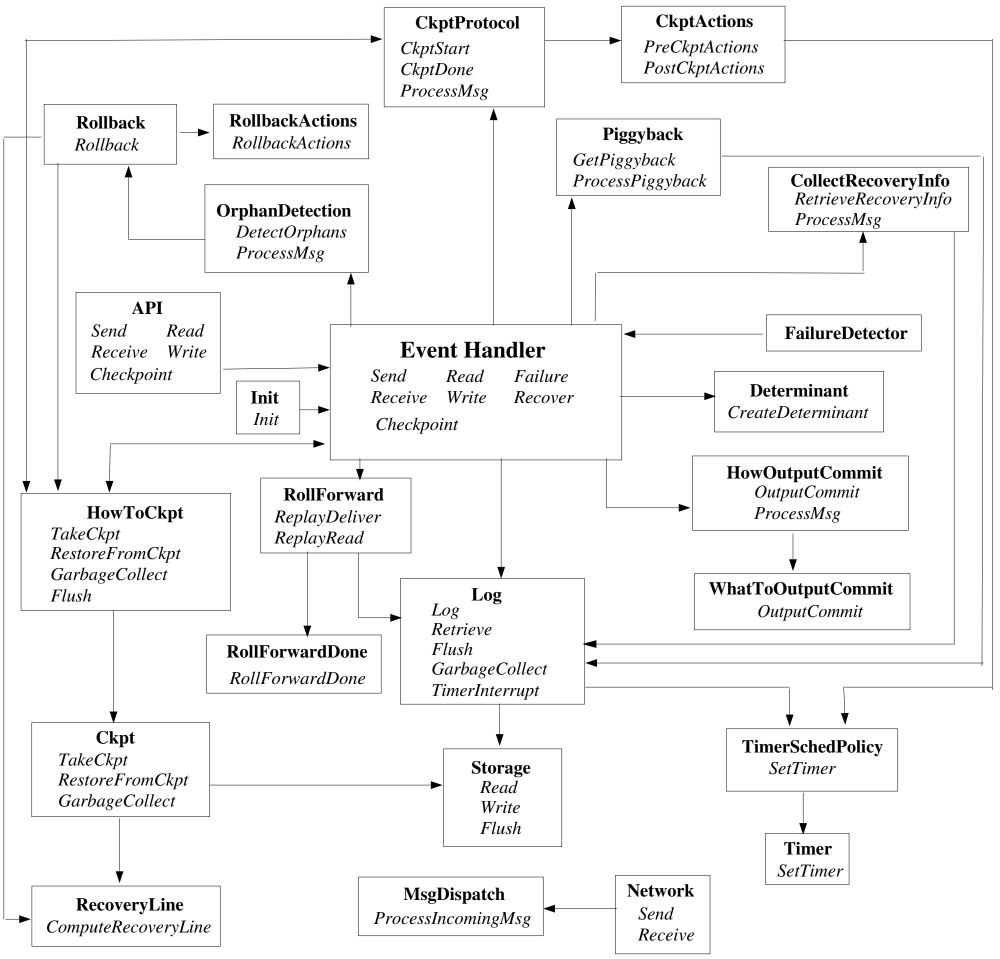


Figure 2.5: The architecture of Egida.

Figure 2.5 shows the function call graph for the modules introduced in Section 2.3.2. To illustrate the interactions among Egida's module, we present two examples that show how this dependency graph is traversed in response to specific events.

Example 1 *Handling a Send event for a causal logging protocol* (see [3])*:*

The API module intercepts the Send event and invokes the corresponding handler in EventHandler. The handler in this example performs four tasks: (1) it saves in a volatile log kept by sender the content of the message being sent; (2) it obtains piggybacking information from the checkpointing protocol; (3) it obtains piggybacking information from the logging protocol—in the case of causal logging, this results in retrieving a set of non-stable determinants from the log; (4) it appends the piggybacking information on the application message and sends the message to the destination.

1. To save the content of the message in the sender's volatile log, the handler invokes the *Log* method of LogEventInfo. *Log* in turn invokes the *Write* method of Storage.

2. To obtain the piggyback information from the checkpointing protocol, the handler invokes the *GetPiggyback* method of PiggybackCkpt module.

3. To obtain the determinants to piggyback, the handler invokes the *GetPiggyback* method of PiggybackLog module. This method identifies the non-stable determinants and invokes the *Retrieve* method of LogDeterminant. *Retrieve* in turn invokes the *Read* method of Storage. In causal logging, the instance of Storage for determinants is the VolatileStorage module.

   These methods return the requested determinants which are then passed up through the call chain back to *GetPiggyback*.

4. The event handler then appends the piggyback information to the application message and invokes the Network module's *Send* method. ■

Example 2 *Handling a Send event to the external environment for the optimistic protocol of Damani and Garg* (see [20])*:*

The API module intercepts the send event and invokes the corresponding handler in EventHandler. The handler in this example invokes the output commit procedure prior to sending the message. The output commit procedure in turn performs two tasks: (1) it flushes to stable storage the determinants of all the non-deterministic events executed by the process prior to the send to the external environment, thereby making them recoverable; and (2) it initiates a co-ordinated output commit operation to ensure that all the non-deterministic events that causally precede the send are recoverable.

1. To perform output commit, the handler invokes the *OutputCommit* method of HowToOutputCommit. The *OutputCommit* method performs the following two tasks:

    (a) To perform a local output commit, *OutputCommit* invokes the *OutputCommit* method of WhatToOutputCommit which then invokes the *Flush* method of LogDeterminant. *Flush* then invokes the *Flush* method of Storage. The specific instantiation of the Storage module used by this protocol is VolatileStorage. To save the determinants on stable storage, VolatileStorage module's *Flush* method invokes the *Write* method exported by StableStorage.

    (b) To perform a co-ordinated output commit, *OutputCommit* broadcasts a message to the remaining processes requesting an output commit and blocks for an acknowledgment message from each process. On receiving the output-commit message, the Network module invokes the MsgDispatch module's *ProcessIncomingMsg*, which in turn invokes the *ProcessMsg* method of HowToOutputCommit. The *ProcessMsg* method invokes the same sequence of methods as in (a) and then sends back an acknowledgment.

2. The handler sends the application message to the external environment. ■

2.3.4 Synthesizing Protocols through Module Composition

Egida allows the co-existence of multiple implementations for each of the modules introduced in Section 2.3.2. Hence, to synthesize a protocol, one must select a specific implementation of each module. Egida maintains a binding between the values for the variables on the left hand side of Figure 2.2 and their corresponding implementations. Therefore, synthesizing a protocol requires processing the specification along with this binding information to initialize the modules to their appropriate implementations. It is easy to extend

Egida to recognize new implementations of modules; as more implementations of a module functionality become available, the specification language can be extended by simply (1) adding a new value to the right hand side of the corresponding variable in Figure 2.2, and (2) registering with Egida the binding between the new value and its corresponding implementation.

2.4 Related Work

The literature contains several systems that have been designed to provide transparent fault-tolerance using rollback-recovery. They include libckpt [60], Fail-safe PVM [48], MIST [16], Co-check [77], Manetho [25] and libft [34]. As opposed to Egida, however, these systems are designed to support only a specific checkpointing or message-logging protocol.

Egida's emphasis on code re-usability and extensibility is also present in the recently-introduced OTEC simulator [64]. OTEC uses an object-oriented design for providing some degree of code re-usability. Though OTEC can be used to compare the performance of different rollback-recovery protocols, the comparison can be done only through simulations (for example, varying message rates and checkpoint sizes). The run-time system for rollback-recovery that is closest to Egida in terms of extensibility is RENEW [57]. RENEW provides the basic send and receive interfaces which can be used to implement any rollback-recovery protocol. The architecture however does not facilitate re-using components across protocol implementations. Egida's uniqueness stems from allowing implementations of rollback-recovery protocols to be synthesized from reusable components. The specification language exported by Egida identifies key properties of rollback recovery protocols, simplifying our understanding, while hiding the tedium of combining the various properties to implement a complete protocol.

Egida's approach of configuring protocols from re-usable components is similar to the one used in other systems—for instance, in Adaptive [69] and xkernel [35] for networking protocols, in Horus [67] and Cactus [32] for distributed computing and group membership protocols, and in COMERA [84] and Quarterware [75] for communications middleware. However, to our knowledge, Egida is the first application of this approach to rollback recovery protocols.

2.5 Summary

In this chapter, we presented the design of *Egida*, a toolkit for supporting transparent rollback-recovery. To design Egida, we developed a single unifying framework based on handling non-deterministic events in a process' execution. This framework allows us to characterize rollback-recovery protocols as event-driven programs which differ in their response to a same set of "relevant" events. We identified five classes of events that are relevant to all protocols and developed a simple specification language for expressing rollback-recovery protocols in terms of these events. Egida's architecture is built around a set of "building blocks" that provide the functionality for handling the relevant events. From a specification, Egida can synthesize an implementation of the specified protocol by gluing together the appropriate objects from an available library of building blocks. Our approach has several advantages. First, it promotes extensibility and flexibility by allowing multiple implementations of each of the building blocks. Second, it facilitates rapid implementation of rollback recovery protocols with minimal programming effort by gluing together objects from the available library of building blocks. This enables application developers to experiment with different protocols and then use the one that closely matches the requirements of their application. Finally, Egida enables designers of fault-tolerance protocols to develop new rollback-recovery protocols by combining different implementations of the building blocks in novel ways.

Chapter 3

Implementation and Experimental Setting

We have implemented the modules introduced in Section 2.3 as C++ classes. These classes correspond to the values on the right hand side of Figure 2.2. To make Egida available to a large class of applications, we have integrated Egida with MPICH [29], a freely available, portable implementation of the MPI (Message Passing Interface) standard 1.1. In Section 3.1 we provide a brief overview of MPICH's architecture and explain how MPICH and Egida are integrated. In the subsequent chapters of this dissertation, we use Egida as a testbed to evaluate the performance of rollback-recovery protocols. The experimental setting used for the performance studies is described in Section 3.2.

3.1 Integration with MPICH

MPICH has a two-layer architecture: the upper layer exports MPI's application programming interface (API), while the lower layer contains platform-specific message-passing libraries. For our testbed environment consisting of a network of workstations connected by an Ethernet, MPICH's lower layer is implemented using the p4 library [15].

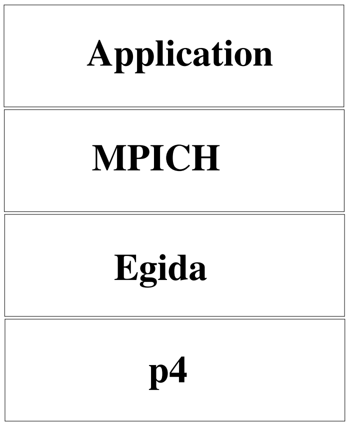


Figure 3.1: Egida integrated with MPICH

To integrate Egida with MPICH, we replaced in the MPICH's upper layer all the send and receive calls to p4 with corresponding calls to Egida's API; modules in Egida in turn invoke the message passing operations of p4. We had to make two modifications to p4. First, we enhanced p4 to enable transfer of data from non-contiguous buffers. This was necessary to avoid a potential data copy that may occur when the information piggybacked by Egida is prepended to an MPICH message. Second, we modified p4 to handle socket errors that occur whenever processes fail and to allow a recovering process to establish socket connections with the surviving processes. Figure 3.1 shows the resulting system structure. Our implementation provides transparent fault-tolerance to MPI applications—only a simple re-link of the application with Egida and our modified MPICH library is necessary. Finally, for detecting failures and automatically re-starting crashed processes, we have implemented a watchdog [34] for each application process.

3.2 Experimental Setting

The testbed environment in which we use Egida to evaluate the performance of rollback-recovery protocols consists of a set of 300-MHz Pentium-II based workstations running Solaris 2.7 and connected by a 100Mbps Ethernet. Each workstation has two processors, 512MB of RAM, and a 4GB local disk. Unless noted otherwise, applications were run using four processes with one process per machine.

All results are reported with a 95% confidence interval and the variance is within 1% of the mean. The number of repetitions (n) required to obtain this accuracy level is computed using a two-step procedure. We first measure the sample mean and standard deviation by repeating each experiment five times. Next, we compute n using the formula [36]:

$$n = \left(\frac{100\ z\ s}{r\ \overline{x}}\right)^2 \tag{3.1}$$

where z is the normal variate of the desired confidence level, s is the sample standard deviation, r is the desired accuracy level, and $\overline{x}$ is the sample mean. In our case, the values of z and r are respectively 1.96 and 1.

The applications that we use for our experiments are from the NPB2.3 benchmark suite developed by NASA's Numerical Aerodynamic Simulation program [17]. These benchmarks are derived from computational fluid dynamics codes and typify the kind of applications that have traditionally benefited from checkpointing and message-logging protocols. Each benchmark is characterized by a *class*, which is one of S, A, B, and C, and the class defines the problem size (for instance, the size of an input matrix over which a computation is to be performed). In the NPB 2.3 suite, for each benchmark the problem size (and in turn, the computation time) increases in the order of S, A, B, and C. The running times of the benchmarks used in the study, the per-process average message rate and size, and the communication patterns of processes are shown in Table 3.1 and Figure 3.2.

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

Table 3.1: Characteristics of the benchmarks used in the experiments.

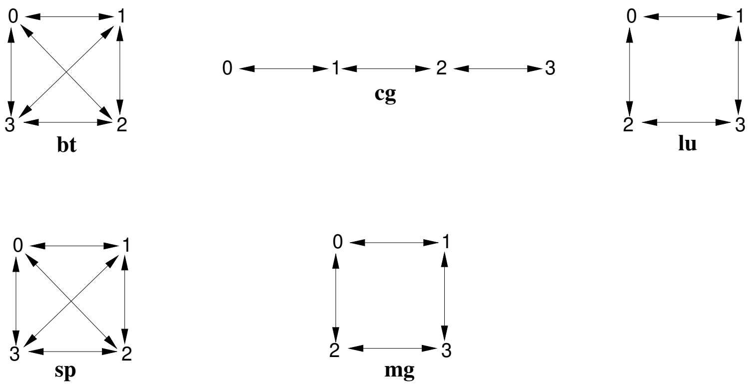


Figure 3.2: Communication patterns for our benchmark applications

Chapter 4

Evaluating Recovery Performance of Message-logging Protocols

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. While previous experimental studies have measured the overhead imposed by each of the three styles of message-logging during failure-free executions [26, 37], 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 back, and then roll forward all orphan processes. On the other hand, recent optimistic protocols [20] 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, [20, 37, 78]), 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 [20] 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.

To put all of these qualitative arguments into perspective, we use Egida to provide the first experimental study of the recovery performance of pessimistic, optimistic, and causal protocols. In the first part of the chapter, 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. In this study, we make two main contributions.

1. We show 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, we propose a simple way to compute a lower bound for the roll-forward time, and we identify the characteristics of a message logging protocol that allow it to approach this lower bound during recovery.

2. We show 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.

This 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 to be slow during recovery (as in sender based pessimistic and causal protocols) or incur a substantial overhead during failure-free execution (as in receiver-based pessimistic protocols). To address this tradeoff, in the second part of the chapter 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 2% of causal logging during failure-free execution, and within 2% of receiver-based logging during recovery.

The rest of this chapter is organized as follows. In Section 4.1, we describe the message logging protocols used for the study. We describe our experimental methodology, and the metrics for our evaluation in Section 4.2. The experimental analysis of the recovery costs for the pessimistic, optimistic, and causal logging protocols is presented in Sections 4.3 and 4.4. Section 4.5 introduces and evaluates hybrid logging protocols. Finally, Section 4.6 summarizes the lessons learnt from this study.

4.1 Protocols Studied

We use Egida to synthesize implementations of 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* [39]: 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 [1]. This protocol is similar to the one described in [79].

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 literature, we have implemented the protocol described in [20]. This protocol, in addition to tolerating an arbitrary number of failures and preventing the domino-effect, implements a singularly efficient method for detecting orphans processes. In this protocol, causal dependencies are tracked using vector clocks [53]. 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 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 vec-

[1]Some sender-based pessimistic protocols keep both determinants and message contents at the senders [39, 41]. We have not used these protocols for the study because they can only tolerate at most two concurrent failures.

tor 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 Egida, we employ two techniques to improve the performance of optimistic protocols. First, 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 is a modification of the pseudo-code presented in [20] and it has the effect of allowing 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 4.3). Second, a process forks 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 [5]. 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 [6, 25]. Recovery of a failed process proceeds in two phases. In the first phase, the process obtains from the logs of the remaining processes (1) its determinants and (2) 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 pre-crash state. To handle multiple concurrent failures, we implemented a protocol

/* comment: ⟨non-det-event-stmt⟩ */
receive:
determinant : {source, sesn, dest, desn}
Log determinant on local disk
Log message on local disk

/*comment: ⟨dep-gen-event-stmt⟩ */
send:
Piggyback : NULL
Log message on
volatile memory of self

/* comment: ⟨ckpt-stmt⟩ */
Checkpoint : independent, asynchronous
on local disk

/* comment: ⟨recovery-stmt⟩ */
Recovery :
Protocol to retrieve determinants : distributed
Protocol to detect orphans : NULL

(a)

/* comment: ⟨non-det-event-stmt⟩ */
receive:
determinant : {source, sesn, dest, desn}
Log determinant on local disk

/*comment: ⟨dep-gen-event-stmt⟩ */
send:
Piggyback : NULL
Log message on
volatile memory of self

/* comment: ⟨ckpt-stmt⟩ */
Checkpoint : independent, asynchronous
on local disk

/* comment: ⟨recovery-stmt⟩ */
Recovery :
Protocol to retrieve determinants : distributed
Protocol to detect orphans : NULL

(b)

Figure 4.1: Specification of (a) Receiver-based pessimistic message logging protocol and (b) Strom, Bacon, and Yemini's sender-based pessimistic protocol.

that recovers crashed processes without blocking operational processes [22]. 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.

Figures 4.1 and 4.2 show the specification of these protocols.

4.2 Experimental Methodology and 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

```
/* comment: ⟨non-det-event-stmt⟩ */
receive:
    determinant : {source, sesn, dest, desn}
    Log determinant on volatile memory of self,
        local disk
      Flush : periodic
    Log message on volatile memory of self,
        local disk
      Flush : periodic
/*comment: ⟨dep-gen-event-stmt⟩ */
send:
    Piggyback : vector clock
    Log message on
      volatile memory of self
/* comment: ⟨ckpt-stmt⟩ */
Checkpoint : independent, asynchronous
    on local disk
/*comment: ⟨recovery-stmt⟩ */
Protocol to retrieve determinants :
    distributed
Protocol to detect orphans :
    broadcast logical clock
On rollback : Flush determinants, messages
```

(a)

```
/* comment: ⟨non-det-event-stmt⟩ */
receive:
    determinant : {source, sesn, dest, desn}
    Log determinant on
      volatile memory of self
/*comment: ⟨dep-gen-event-stmt⟩ */
send:
    Piggyback : determinants
    Log message on
      volatile memory of self
/* comment: ⟨ckpt-stmt⟩ */
Checkpoint : independent, asynchronous
    on local disk
/*comment: ⟨recovery-stmt⟩ */
Protocol to retrieve determinants :
    centralized
Protocol to detect orphans :
    NULL
```

(b)

Figure 4.2: Specification of (a) Damani-Garg's optimistic message logging protocol and (b) Alvisi-Marzullo's Π_{det} causal message logging protocol.

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.

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 performances 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 running each application for its pre-computed 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 4.2).

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.

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 4.3.4.

4.3 Experimental Results

Before we proceed to analyze in Figure 4.4 and Table 4.1 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 using Table 4.1, which shows the result of our experiments when $1 \leq f < 4$, $n = 4$, and t is chosen half-way 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 T_{rec}. For the class of applications we have used, roll forward contributes more than 90% of T_{rec}, and it never contributes less than 80%.

- 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 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 4.1. Using a 100Mbps Ethernet, with a peak throughput of about 25Mbps, it takes about 25 seconds to transfer to the recovering process the 80MB of data it needs during recovery. In contrast, retrieving the same information from local file system (cache or local disk) requires about 6 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 4.1 shows also that for causal and sender-based pessimistic protocols T_{acq} changes with f: we discuss these effects in Section 4.3.1.

- 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 $f = 1$. Tables 4.1 and 4.3 show that for this application roll-forward for the causal protocol takes about 23% less time than normal execution, while for the receiver-based pessimistic protocol it is about 27% less.

  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 4.3 shows the basic computation time for the three benchmark applications, computed as the difference between the time t at which the failure is induced [2] and the *communication time*, i.e., the amount of time spent by the application in performing send and receive operations.

[2] Recall from the discussion in Section 4.2 that t refers to an execution with no fault-tolerance overhead

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 pessimistic receiver-based 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 during roll-forward and, as Table 4.4 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 4.4 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 is offset from the replay phase of the recovering process by $T_{chk} + T_{acq}$. In our applications, this offset translates in turn in an equivalent blocking that a recovering process incurs after having replayed its logs, while it waits for the orphans to finish the replay of their logs. This happens 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 as well as an orphan, in Table 4.4 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. This has two consequences. First, reconstructing these logs during recovery incurs overheads because of (1) performing memory-to-memory copies and (2) the cost of memory allocation. Table 4.4 shows that for cg, where the size volatile log of sent messages is about 80MB, the overhead of reconstruction 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 4.3.1.

- 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 4.1 shows that in the case of cg the value of T_{rec} for $f = 1$ for the optimized protocol is 161.2 seconds, compared with the value of 282.6 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.

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

Table 4.1: T_{rec} as a function of f, where $1 \leq f < 4$, $n = 4$, and $t \approx 3$min.

Application	Failure Iteration	Pessimistic Receiver-based (sec.)	Pessimistic Sender-based (sec.)	Optimistic (sec.)	Causal (sec.)
bt	16	225.4	197.4	185.2	184.3
cg	9	362.5	215.6	196.3	194.1
lu	42	460.3	310.7	197.2	201.5
sp	70	350.7	240.4	192.8	198.0
mg	17	390.8	224.3	195.3	202.8

Table 4.2: "Wall clock" time at which a failure is induced for each benchmark application. 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.

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

Table 4.3: The values of t, communication time and basic computation time for the benchmark applications.

Application	Pessimistic Receiver-based (sec.)	Pessimistic Sender-based (sec.)	Optimistic (sec.)	Causal (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

Table 4.4: Overhead imposed by protocols during the roll-forward phase of recovery.

4.3.1 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 those lost 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 both the sender-based pessimistic and the causal protocols (see Figure 4.3). This result surprised us: before starting this study, we held the un-substantiated belief 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 4.1 and Table 4.4 shows that our experimental results support this interpretation.

- When $f > 1$, it is still guaranteed that the determinants of all messages are available at the beginning of the roll-forward phase. However, this guarantee does not extend 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. Figure 4.3 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 for bt and lu because they have a much smaller communication time than cg (see Tables 4.1 and 4.3).

- In causal protocols, T_{acq} increases significantly when there are multiple concurrent failures, as the overhead of the algorithm [22] 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 [54], the values of T_{acq} in the causal and sender-based pessimistic protocols would be similar.

4.3.2 Changing the Time of Failure

Figure 4.4(a) 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

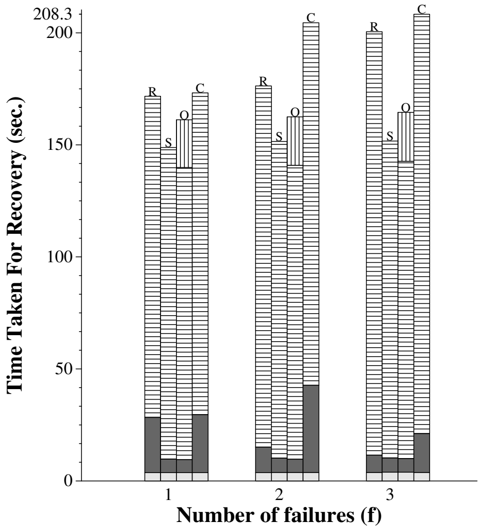


Figure 4.3: T_{rec} as a function of f, where $1 \leq f < 4$, for cg, $n = 4$. For each value of f, we show four bars: receiver-based pessimistic (R), sender-based pessimistic (S), optimistic (O), and causal (C).

the logs are flushed to stable storage, its values does not change with t. Consequently, the contribution of $T_{rollbck}$ to T_{rec} is proportionally reduced.

4.3.3 Changing the Number of Processes

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}. Figure 4.4(b) 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}.

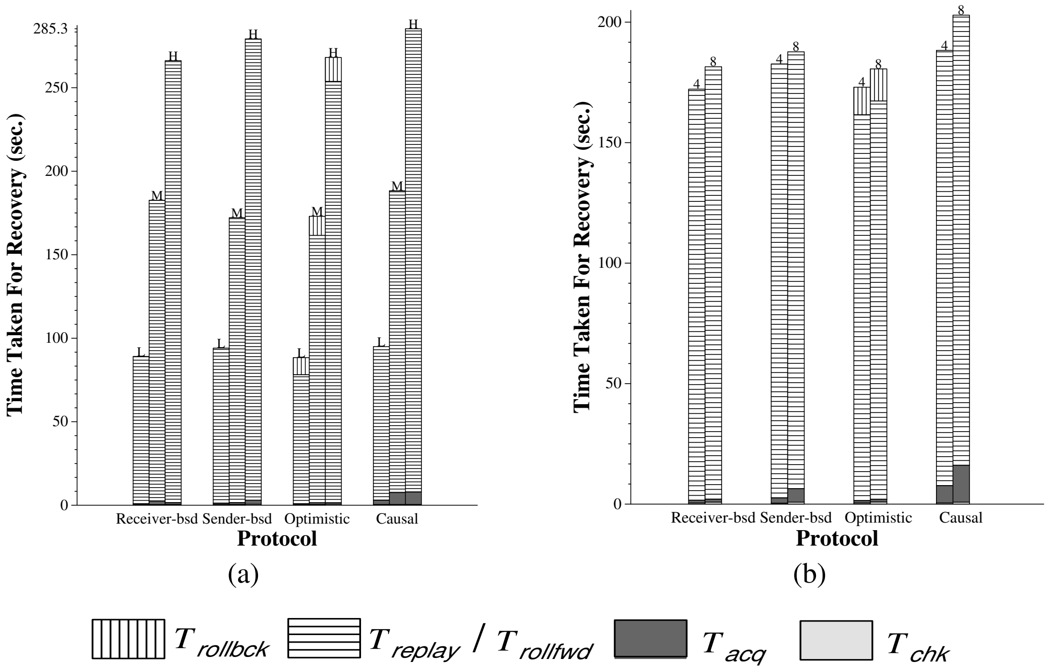


Figure 4.4: (a) Effect on the cost of recovery for lu ($n = 4, f = 3$) of inducing failure after approximately a fourth (L), half (M), and three-fourth (H) of the interval between successive checkpoints. (b) Cost of recovery for lu with 4 and 8 processes, $f = 3$, and $t \approx$ 3min.

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	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					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	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

Table 4.5: 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.

4.3.4 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. Rather, stable storage will have to be implemented on a highly available, remote file system, such as NFS. This implementation of stable storage has two effects on the overall recovery time:

1. For all protocols, to restore a process state using a checkpoint, retrieving the corresponding file from the remote file server involves a network transfer. The cost of the transfer depends on the size of the checkpoint file. In addition, T_{chk} is affected by the load on the file server. As f increases, multiple recovering processes may try to retrieve their checkpoints and logs from the file server concurrently. These requests will be sequentialized at the network interface of the file server, adding to the time necessary to retrieve this data.

2. The cost of acquiring the logs from stable storage will similarly increase for the receiver-based pessimistic and optimistic protocols. Since the sender-based pessimistic protocol only logs determinants to the file server, T_{acq} may not be affected substantially.

To demonstrate these effects, we perform a set of experiments in which the recovering process is re-started on a machine different from the one on which its crashed counterpart was executing prior to failure. For this set of experiments, we synthesize the protocols by modifying the implementation of stable storage from "local disk" to "NFS disk" in the specification shown in Figures 4.1 and 4.2. The results of our experiments are shown in Table 4.5. As expected, (1) for all applications the values of T_{chk} are higher than their corresponding values shown in Table 4.1 and (2) in many cases, T_{chk} doubles as f changes from 1 to 3. The results also show that T_{acq} similarly increases for the receiver-based pessimistic and optimistic protocols. Note that, in Table 4.1, in many cases, T_{acq} in receiver-based pessmistic protocols is higher than the corresponding value of sender-based pessimistic and causal protocols. Consequently, the percentage difference between the T_{rec} values of receiver-based pessimistic protocol and sender-based pessimistic/causal protocol is less when compared to the corresponding difference in Table 4.1, In addition to the increase in T_{chk} and T_{acq}, we also expected that $T_{rollbck}$ in optimistic protocols to increase. This is because when an orphan rolls back, it also has to retrieve its checkpoint and log files from stable storage. The results suggest, for some applications (e.g., bt, lu, sp) the orphans inadvertantly benefit from caching employed by the file system. Since an orphan after a rollback executes on the same machine it was executing prior to the rollback, the information needed for replay can be retrieved from the local cache without incurring any network transfer overheads. Finally, when a recovering process rolls forward and replays messages, it does not access stable storage and hence, $T_{rollfwd}$ in pessimisitic and causal protocols and T_{replay} in optimistic protocols is unaffected by the change in failure model.

4.4 Recovery Performance in the Presence of Load Imbalance Across Workstations

Our benchmark applications partition the overall computational task into several roughly equally-sized components. Each application process executes one such component and exchanges messages with other processes to share intermediate results and to synchronize with them its execution. In Section 4.3, 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 Tables 3.1 and 4.3).

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 an unloaded homogeneous cluster of workstations to derive a new execution trace. Using this new trace, we define a synthetic application that simulates the behaviour of the original application on an 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:

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 4.6 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	$[\mu-\sigma, 1.5*\mu]$			$[\mu-\sigma, 2*\mu]$		
	t (sec.)	**Communication Time** (sec.)	**Basic Computation Time** (sec.)	t (sec.)	**Communication Time** (sec.)	**Basic Computation Time** (sec.)
bt	250.6	38.3	212.3	317.3	72.0	245.3
cg	274.5	55.9	218.6	362.7	82.0	280.7
lu	288.6	63.4	225.2	342.3	87.7	254.6

Table 4.6: The values of t, communication time and basic computation time for the synthetic applications.

synthetic application for the same number of iterations—shown in Table 4.2—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 4.6 are higher than the corresponding values in Table 4.3.

Table 4.6 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% larger (see Table 4.3). 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 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 applies also 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

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	241.8	297.5
	3	254.9	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

Table 4.7: Recovery performance of causal protocol for synthetic applications ($1 \leq f < 4$ and $n = 4$)

causal protocols, on the other hand, we have seen in Section 4.3.1 that for $f > 1$ less messages are available at the beginning of the roll-forward phase, causing roll forward to proceed more and more at the speed of failure-free execution and $T_{rollfwd}$ to depart from the basic computation time—what we called the stop-and-go effect.

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 5% (see Table 4.1) from $f = 1$ to $f = 3$, Table 4.7 show that for their synthetic counterparts the difference grows to about 30%. Similar results hold also for the sender-based pessimistic protocol.

4.5 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 to be relatively slow during recovery—like sender based pessimistic and causal protocols—or

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 4.8: Failure-free overhead imposed by the various logging protocols

incur a substantial overhead during failure-free execution—like receiver-based pessimistic protocols (see Table 4.8).

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

It is easy to synthesize implementations of hybrid protocols using Egida. Figure 4.5 shows the specification for the hybrid versions of sender-based pessimistic and causal protocols. The specification of the traditional versions of the corresponding protocols (see Figures 4.1(b) and 4.2(b)) is enhanced to include the following lines: (1) in hybrid sender-

/* `comment:` ⟨non-det-event-stmt⟩ */
receive:
  determinant : {source, sesn, dest, desn}
  Log determinant on local disk

  Log message on volatile memory of self,
    local disk
   Flush : periodic

/*`comment:` ⟨dep-gen-event-stmt⟩ */
send:
  Piggyback : NULL
  Log message on
   volatile memory of self

/* `comment:` ⟨ckpt-stmt⟩} */
Checkpoint : independent, asynchronous
  on local disk

/* `comment:` ⟨recovery-stmt⟩ */
Recovery :
  Protocol to retrieve determinants : distributed
  Protocol to detect orphans : NULL

(a)

/* `comment:` ⟨non-det-event-stmt⟩ */
receive:
  determinant : {source, sesn, dest, desn}
  Log determinant on volatile memory of self,
    local disk
   Flush : periodic
  Log message on volatile memory of self,
    local disk
   Flush : periodic

/*`comment:` ⟨dep-gen-event-stmt⟩ */
send:
  Piggyback : determinants
  Log message on
   volatile memory of self

/* `comment:` ⟨ckpt-stmt⟩ */
Checkpoint : independent, asynchronous
  on local disk

/* `comment:` ⟨recovery-stmt⟩ */
Recovery :
  Protocol to retrieve determinants : centralized
  Protocol to detect orphans : NULL

(b)

Figure 4.5: Specification of (a) hybrid sender-based pessimistic message logging protocol and (b) hybrid causal message logging protocol.

based pessimistic logging, it is necessary for a process to log its delivered messages to stable storage and (2) in hybrid causal logging, it is necessary for a process to log determinants and messages it delivers to stable storage.

We have evaluated the performance of hybrid versions of sender-based pessimistic and causal protocols. Table 4.9 shows that hybrid logging dramatically reduces the recovery cost of the sender-based pessimistic and causal protocols, which are now within 2% of the receiver-based protocol, even when $f > 1$ (see Tables 4.1 and 4.9). At the same time, Table 4.10 shows that when hybrid logging is used in place of sender-based logging, the failure-free execution time increases by at most 2%. In practice, hybrid causal protocols are more desirable, because causal logging imposes significantly less overhead during failure-

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

Table 4.9: T_{rec} in hybrid protocols as a function of f, where $1 \leq f < 4$, $n = 4$, and $t \approx 3$min.

free executions than sender-based pessimistic protocols (see Table 4.8).

4.6 Summary

Our results reveal several characteristics of the recovery performance of message-logging protocols. 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 protocol always achieves this lower bound, and hence have the best recovery performance. We also found 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

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%

Table 4.10: Increase in the failure-free overhead imposed by hybrid protocols when compared to their traditional counterparts

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 re-delivered 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 existing message-logging protocols do not simultaneously support fast failure-free execution, fast recovery, and fault-containment. To address this limitation, we proposed a new class of orphan-free protocols—referred to as hybrid protocols. Our evaluation showed that hybrid protocols perform within 2% of causal logging during failure-free execution, and within 2% of receiver-based logging during recovery.

Chapter 5

Evaluation of Communication-Induced Checkpointing Protocols

Communication-induced checkpointing (CIC) protocols have been proposed as a practical candidate for providing low-overhead fault-tolerance using just checkpointing. These protocols allow processes to take checkpoints independently without risking the domino-effect [65]. CIC protocols are believed to have several advantages over other styles of rollback-recovery. For instance, they give processes considerable autonomy in deciding when to take checkpoints thereby allowing a process to checkpoint when saving its state will incur a low overhead [49, 59]. CIC protocols are also believed to scale up well with a larger number of processes since they do not require the processes to participate in a co-ordinated checkpoint. But there is a price to pay for these advantages. First, the protocol-specific information piggybacked on application messages occasionally "induces" processes to take forced checkpoints before they can process the messages. Second, processes have to pay the overhead of piggybacking information on application messages, and they also need to keep several checkpoints on stable storage. These advantages and disadvantages have been so far argued qualitatively. The objective of our performance study is to shed some light on these issues using quantitative metrics drawn from a real system.

To this end, we use Egida to study three CIC protocols—–the original one by Briatico et al. [13], and two recent protocols [9, 30] based on the Z-path theory [56]. We evaluate their performance using two metrics, namely, the overhead imposed on application running time and the average number of forced checkpoints a protocol causes. The second metric is important because forced checkpoints negate the advantage of autonomy of CIC protocols and contribute substantially to the performance and resource overheads. A good CIC protocol therefore tries to limit these forced checkpoints to the extent possible.

The rest of this chapter is organized as follows. A brief overview of the three CIC protocols is presented in Section 5.1. Section 5.2 contains a discussion of implementation issues in CIC protocols. Section 5.3 presents the results of the performance study. Finally, Section 5.4 summarizes the lessons learnt from the evaluation.

5.1 Background

This section reviews the three protocols used in the experiments, along with some necessary definitions. In this chapter, we use $C_{i,j}$ to denote the j^{th} checkpoint of process p_i; and $send_i$ and $deliver_i$ to denote, respectively, the send and deliver events executed by process p_i.

5.1.1 Definitions

Local checkpoints: A process may take a local checkpoint at any time during its execution. Different processes do not coordinate their local checkpoints to form a global consistent checkpoint [18].

Forced checkpoints: To guard against the domino-effect, a CIC protocol piggybacks protocol-specific information to application messages that processes exchange. Each process examines this information and is occasionally forced to take a checkpoint according to the protocol.

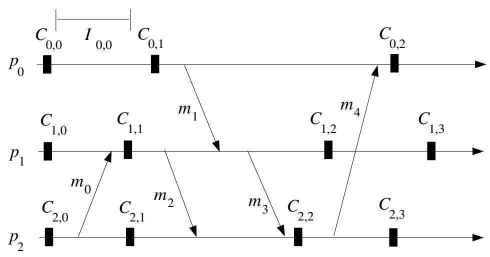


Figure 5.1: A distributed computation.

Useless checkpoints: A useless checkpoint is one that will never be part of a consistent global state [82]. In figure 5.1, any global state that includes checkpoint $C_{2,2}$ is inconsistent and hence, $C_{2,2}$ is a useless checkpoint. Useless checkpoints are not desirable because they do not contribute to the recovery of the system from failures, but they consume resources and cause performance overhead.

Checkpoint intervals: A checkpoint interval is the sequence of events between two consecutive checkpoints in the execution of a process. For example, in Figure 5.1, $I_{0,0}$ corresponds to the checkpoint interval defined by $C_{0,0}$ and $C_{0,1}$.

Message chains: A message chain [9] is a sequence of messages $[m_0, m_1, \ldots, m_z], z \geq 0$ such that $\forall l,\ 0 \leq l < z,\ (\mathit{deliver}_i(m_l) \in I_{i,x}) \wedge (\mathit{send}_i(m_{l+1}) \in I_{i,y}) \wedge ((x = y) \vee (\mathit{deliver}_i(m_l) \rightarrow \mathit{send}_i(m_{l+1})))$. For example, in Figure 5.1, $[m_1, m_2, m_4]$ is a message chain.

Causal message chains: A message chain $[m_0, m_1, \ldots, m_z], z \geq 0$ is *causal* [9] if $\forall l,\ 0 \leq l < z,\ \mathit{deliver}_i(m_l) \rightarrow\ \mathit{send}_i(m_{l+1})$; otherwise, the chain is non-causal. For example, in Figure 5.1, $[m_1, m_3]$ is a causal message chain and $[m_1, m_2]$ is a non-causal message chain. Note that a message chain consisting of a single message is causal [9]. We use the symbol λ to denote a causal message chain and write $\lambda.\mathit{first}$ and $\lambda.\mathit{last}$ to denote, respectively, the first and last message in the causal chain λ.

We use $M(C_{i,x}, p_k)$ to denote the set of causal chains λ such that $C_{i,x} \rightarrow \lambda.\mathit{first}$ and the receiver of $\lambda.\mathit{last}$ is p_k. λ is *prime* [9] in $M(C_{i,x}, p_k)$ if and only if there does not exist $\lambda' \in M(C_{i,x}, p_k)$ such that $\mathit{deliver}_k(\lambda'.\mathit{last}) \rightarrow \mathit{deliver}_k(\lambda.\mathit{last})$.

Z-paths: A Z-path (zigzag path) is a sequence of messages that define the relation $\stackrel{z}{\rightarrow}$ between a pair of checkpoints [56]. We write $C_{i,m} \stackrel{z}{\rightarrow} C_{j,n}$ if and only if one of the following two conditions hold [56]:

1. $m < n$ and $i = j$; or
2. There exists a message chain $[m_0, m_1, \ldots, m_z]$, $z \geq 0$, such that:
   (a) $C_{i,m} \rightarrow$ *send*$_i(m_0)$;
   (b) *deliver*$_j(m_z) \rightarrow C_{j,n}$.

For example, in Figure 5.1, $[m_1, m_2]$ and $[m_1, m_3]$ are Z-paths and hence, in either case, $C_{0,1} \stackrel{z}{\rightarrow} C_{2,2}$.

Z-cycles: A Z-cycle is a Z-path that begins and ends with the same checkpoint [56]. In Figure 5.1, the Z-path $[m_4, m_1, m_3]$ is a Z-cycle that involves checkpoint $C_{2,2}$.

5.1.2 Z-cycles and CIC

CIC protocols do not take useless checkpoints. These protocols exploit the observation that the creation of useless checkpoints depends on the occurrence of specific patterns in which processes communicate and take checkpoints [31]. Informally, these protocols recognize potentially dangerous patterns and break them before they occur. This intuition has been formalized in an elegant theory based on Z-cycles. A key result in this theory is that if a Z-path exists between a pair of checkpoints, then both the checkpoints cannot belong to the same consistent global state [31, 56]. It follows from this result that a checkpoint involved in a Z-cycle cannot belong to any consistent global state [31, 56]. Hence, to avoid useless checkpoints it suffices that no Z-path ever becomes a Z-cycle. Enforcing the no-Z-cycle ($\mathcal{NZC}$) condition may require that a process save forced checkpoints in addition to its local checkpoints. There are two approaches to avoiding Z-cycles. The first approach uses a function that associates a timestamp with each checkpoint. The protocol enforces, through forced checkpoints if necessary, the following two rules [31]:

Rule 1: If there are two checkpoints $C_{i,m}$ and $C_{j,n}$ such that $C_{i,m} \xrightarrow{z} C_{j,n}$, then $ts(C_{j,n}) \geq ts(C_{i,m})$, where $ts(C)$ is the timestamp associated with checkpoint C.

Rule 2: Consecutive local checkpoints of a process have increasing timestamps.

The second approach relies instead on preventing the formation of specific checkpoint and communication patterns that may lead to the creation of a Z-cycle. Protocols that follow this approach do not adopt a specific function for associating timestamps with checkpoints. However, for these protocols there always exists an equivalent time-stamping function that would cause the same forced checkpoints [31].

In what follows, we describe the three CIC protocols used in our study in terms of these two approaches. Of these protocols, the first two use the first approach and the third uses the second approach.

5.1.3 Briatico, Ciuffoletti and Simoncini (BCS)

In BCS, each process p_i maintains a logical clock lc_i that operates as p_i's checkpoint timestamp. To trace dependencies between checkpoints, p_i piggybacks on every out-going message m a copy of the current value of lc_i. We denote this value as $m.lc$. The timestamp is an integer variable with an initial value 0 and is incremented according to the following function:

1. lc_i increases by 1 whenever p_i takes a local checkpoint.

2. Whenever p_i receives a message m, it compares lc_i with $m.lc$. If $m.lc > lc_i$, then p_i sets lc_i to the value of $m.lc$ and takes a forced checkpoint before processing the message.

Although this protocol was developed before the Z-path framework, it is easy to see that the timestamping function used in BCS follows Rule 1 and Rule 2 mentioned above: To prove Rule 1, suppose that there are two checkpoints $C_{i,m}$ and $C_{j,n}$ such that $C_{i,m} \xrightarrow{z} C_{j,n}$.

Now, the logical clock piggybacked on each message along the Z-path from $C_{i,m}$ to $C_{j,n}$ is at least $ts(C_{i,m})$. This is because the Z-path begins with a message sent by p_i after $C_{i,m}$ and the logical clock piggybacked on that message by p_i is at least $ts(C_{i,m})$; furthermore, the second rule ensures that the logical clock of each process that delivers a message along the Z-path is also at least $ts(C_{i,m})$. Hence, when p_j takes $C_{j,n}$, $ts(C_{j,n}) = lc_j \geq ts(C_{i,m})$. To satisfy Rule 2, note that lc_i increases whenever a process takes a checkpoint (forced or local). The set of checkpoints having the same timestamps in different processes is guaranteed to be a consistent global state. Therefore, this protocol guarantees that there is always a recovery line corresponding to the lowest timestamp in the system, and the domino effect is prevented.

5.1.4 Hélary, Mostefaoui, Netzer and Raynal (HMNR)

The HMNR protocol uses the observation that if checkpoints' timestamps always increase along a Z-path (as opposed to simply non-decreasing, as required by Rule 1 above), then no Z-cycle can ever form [30]. Based on this observation, Hélary et al. design functions that minimize the number of forced checkpoints. They begin with a simple protocol which would require each process to maintain a logical clock, as in BCS, and to apply the following rules [30]:

1. lc_i increases by 1 whenever p_i takes a local or forced checkpoint.
2. Whenever p_i sends a message m, it piggybacks on m a copy of lc_i; as before, we denote this value by $m.lc$.
3. Whenever p_i receives a message m, it compares lc_i with $m.lc$. If $m.lc > lc_i$, then p_i sets lc_i to the value of $m.lc$.

Then, they refine this protocol using more sophisticated observations and requiring that processes append on out-going messages more information, such as the logical clocks of all processes. Figure 5.2, adapted from [30], shows how this additional information can

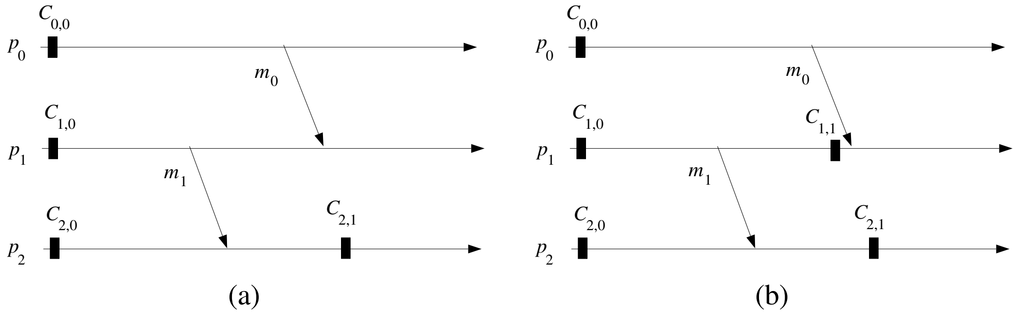


Figure 5.2: To checkpoint or not to checkpoint?

be used to decide when to take a checkpoint. In the figure, $C_{0,0}$ is the first checkpoint of p_0 before sending m_0 and $C_{2,1}$ is the first checkpoint of p_2 after delivering m_1. Since the sending of m_1 and the delivery of m_0 belong to the same checkpoint interval of p_1, $[m_0, m_1]$ is a Z-path between $C_{0,0}$ and $C_{2,1}$. When process p_1 receives message m_0, there are two cases to consider:

1. $m_0.lc \leq m_1.lc$. In this case, there is no need for a forced checkpoint because timestamps increase along the Z-path: $ts(C_{0,0}) \leq m_0.lc \leq m_1.lc < ts(C_{2,1})$. By Rule 1 (see Section 5.1.2), no Z-cycle is created.

2. $m_0.lc > m_1.lc$. In this case, the delivery of m_1 may create a Z-path along which timestamps do not increase. To avoid this risk, p_1 can simply break the Z-path by taking a forced checkpoint before delivering m_1 (as shown in Figure 5.2(b))[1]. However, p_1 can use the information piggybacked on m_1 (i.e., the logical clocks of all processes) to decide if a forced checkpoint is really necessary. In the figure, if p_1 knows that the value of p_2's local clock is at least x when it is about to deliver m_0, then p_1 does not need to take a forced checkpoint if $ts(C_{0,0}) \leq ts(m_1) \leq x < ts(C_{2,1})$. In this scenario, timestamps increase along the Z-path, and by Rule 1 (see Section 5.1.2), no Z-cycle will be created.

[1]It is interesting to note that the BCS protocol uses exactly this approach.

In [30], a protocol using this analysis is presented first; it is then refined to develop several CIC algorithms [30]. For this study, we have used the most sophisticated among them, which reduces the number of forced checkpoints as much as possible.

5.1.5 Baldoni, Quaglia and Ciciani (BQC)

The BQC protocol does not prevent forced checkpoints by using an explicit function to timestamp checkpoints [9]. Rather, this protocol enforces $\mathcal{NZC}$ by preventing the formation of checkpoint and communication patterns that may result in the creation of a Z-cycle. In particular, BQC prevents the creation of a suspect Z-cycle [9].

Formally, a suspect Z-cycle [9] involves $C_{i,x}$ if there exists a checkpoint and communication pattern with messages m_0 and m_1, checkpoint intervals $I_{j,y}$ and $I_{k,z}$, and causal message chain λ, satisfying the following rules:

1. $\mathit{send}_j(m_0) \in I_{j,y}$
2. $\exists v > 0 : \mathit{deliver}_i(m_0) \in I_{i,x-v}$
3. $\exists v \geq 0 : \mathit{send}_i(\lambda.\mathit{first}) \in I_{i,x+v}$
4. $(\mathit{deliver}_k(\lambda.\mathit{last}) \in I_{k,z}) \wedge (\mathit{send}_k(m_1) \in I_{k,z}) \wedge (\mathit{send}_k(m_1) \rightarrow \ \mathit{deliver}_k(\lambda.\mathit{last}))$
5. λ is prime in $M(C_{i,x}, p_k)$
6. $\nexists e \in I_{j,y+1} \ : \ e \rightarrow \mathit{deliver}_k(\lambda.last)$, where e is a communication event (send or deliver) executed by p_j.

Figure 5.3(a), adapted from [9], shows the structure of a suspect Z-cycle that involves $C_{i,x}$. Figure 5.3(b), also adapted from [9], shows a checkpoint and communication pattern that resembles the structure of a suspect Z-cycle. However, in Figure 5.3(b), there is no suspect Z-cycle involving $C_{i,x}$ because the causal chain λ' contradicts the last rule in the suspect Z-cycle definition.

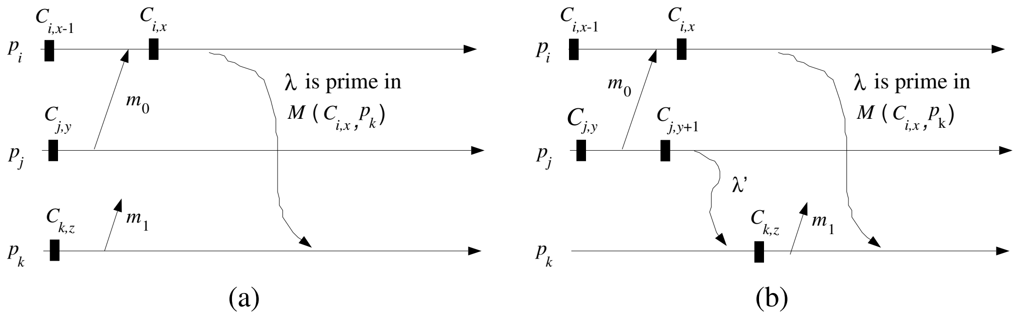


Figure 5.3: A suspect Z-cycle involving checkpoint $C_{i,x}$ is shown in (a); a structure that resembles a suspect Z-cycle, but is not a suspect Z-cycle is shown in (b).

A suspect Z-cycle is not necessarily a Z-cycle. For instance, there is no Z-cycle in Figure 5.3(a) unless there is an actual Z-path starting with m_1 and ending with m_0 between $C_{k,z}$ and $C_{i,x}$. This information may or may not be available to process p_k when it receives λ.*last*. If the information is not available, then p_k opts for safety and takes a forced checkpoint before delivering λ.*last*; otherwise, p_k refrains from taking a forced checkpoint.

The actual protocol presented in the paper [9] propagates $O(n^2)$ values on each application message to help processes detect suspected Z-cycles and suppress them using forced checkpoints. Reference [9] also contains a proof that a protocol that prevents a suspect Z-cycle also satisfies $\mathcal{NZC}$.

5.2 Implementation Issues

An implementation of CIC must deal with several pragmatic issues that are typically left out of protocol specifications. We describe how we resolved these issues for our performance study.

5.2.1 Autonomy in Local Checkpoints

A stated advantage of CIC protocols is that they prevent the domino effect while allowing processes considerable autonomy in deciding when to take local checkpoints. An efficient implementation of CIC must therefore adopt a checkpointing policy that exploits this autonomy and translates it into a benefit. In general, this requires a good understanding of the application and of the execution environment. For example, detailed knowledge of the application may allow processes to take checkpoints when the size of the live variables is small [49, 59].

We have found that either the complexity of the application programs (see Chapter 3) precludes investing the effort in defining a reasonable checkpoint placement policy, or that the application structure does not reveal points within the execution where taking a checkpoint is more advantageous than others. One of the first conclusions that we reached is that even with a deep understanding of the applications' structure, no policy for taking local checkpoints can reasonably be adopted without considering the effect of the forced checkpoints that occur because of communication events. For example, a pre-run analysis may decide on the execution points during which it is most "convenient" to schedule local checkpoints within a particular process. But if this policy ignores the forced checkpoints, then it is possible to schedule a local checkpoint immediately after a forced checkpoint. In this case, taking the local checkpoint will result in additional work and overhead, with no substantial reduction of the amount of work at risk. A more reasonable decision may then be to postpone taking the local checkpoint. In any case, the autonomy of deciding when to take checkpoints seems to be limited by the occurrence of forced checkpoints due to interactions with other processes. Therefore, we have resolved to use a probabilistic distribution to emulate what an autonomous application would do in deciding on where to place the local checkpoints.

5.2.2 Blocking versus Non-blocking Checkpointing in CIC

CIC protocols, to maintain the invariant that no Z-cycles may form and no useless checkpoint is ever taken, require the use of blocking (synchronous) checkpointing. However, the use of blocking checkpointing is known to impose a high performance overhead [24, 60].

Our results re-confirm that, indeed, the overhead is high. The alternative is to then use non-blocking (asynchronous) checkpointing.

Non-blocking checkpointing allows the application to resume computation as soon as possible and to schedule the actual writing of the checkpoint concurrently with the application execution. The result is that saving the checkpoint to stable storage does not become a bottleneck that impedes application progress. However, using non-blocking checkpointing in CIC introduces potential inconsistencies with the specification of the protocols. To understand why, consider the situation where a process p_i takes a non-blocking checkpoint (local or forced) and then sends a message m to another process p_j; assume furthermore that the CIC protocol being used requires p_j to take a forced checkpoint before delivering m. Now, suppose that p_i fails after sending m, but before p_i's checkpoint has been entirely written to stable storage. The failure of p_i makes the checkpoint taken by p_j useless, thereby violating the protocol invariant. The problem is not just cosmetic, because several such communication events may occur while several non-blocking checkpoints are being written to stable storage according to an implementation of non-blocking checkpointing. Therefore, it may be possible for many checkpoints on stable storage to be rendered useless because a failure of some process in the system occurred before one or more of its checkpoints were saved on stable storage. Indeed, in situations like this, Z-cycles do form and useless checkpoints are taken. There are two solutions to this problem:

1. Block any outgoing messages from a process until all its non-blocking checkpoints have been written to stable storage. The messages are buffered and released only when a process receives notification from the checkpointing agent that the checkpoint has been saved. There is a penalty to pay for this modification, but it is preferable to disallowing non-blocking checkpointing altogether. It is interesting to note here that this solution shows how a pragmatic consideration may require a modification in the protocol itself to maintain its invariant.

2. Simply allow these temporary Z-cycles to form, and hope that the checkpoints will be

written before a failure actually occurs. In a sense, this is an optimistic implementation of CIC, which may allow Z-cycles to form temporarily while some checkpoints are being written to stable storage. If the optimistic assumption holds, then the invariant of the protocol is preserved and no useless checkpoints are ever taken. However, if the assumption is violated because of a failure, then some of the checkpoints that would have otherwise be part of the recovery line will have to be discarded. The benefit of this optimistic alternative is that the overhead is small and does not require any modification to the protocol as specified.

In the experimental study, we first show that using blocking checkpointing in CIC leads to poor performance; we then use the second solution because non-blocking checkpointing is known to impose low overhead [24, 60] and also for its simplicity.

5.3 Experiments and Analysis

5.3.1 Methodology and Metrics

The study is based on a combination of experiments using Egida and validated simulations that allow us to study CIC protocols further, under different communication patterns and environments. Figure 5.4 shows the Egida specification for the three protocols used in the study. The performance metrics we report are the number of forced checkpoints that a protocol causes and the performance overhead. In the experiments, to minimize the checkpointing overhead, stable storage for checkpoint files is implemented using the local disk of the workstation on which a process is executing.

/*`comment`: Specifying BCS protocol */

/*`comment`: ⟨dep-gen-event-stmt⟩ */
send:
  Piggyback : checkpoint counter
/* `comment`: ⟨ckpt-stmt⟩ */
Checkpoint : independent, asynchronous on local disk
Implementation : incremental
Scheduling policy : random
On checkpoint : Update checkpoint counter

/*`comment`: Specifying HMNR protocol */

/*`comment`: ⟨dep-gen-event-stmt⟩ */
send:
  Piggyback : checkpoint counter, checkpoint vector, taken, greater
/* `comment`: ⟨ckpt-stmt⟩ */
Checkpoint : independent, asynchronous on local disk
Implementation : incremental
Scheduling policy : random
On checkpoint : Update checkpoint counter, checkpoint vector, taken, greater

---

/*`comment`: Specifying BQC protocol */

/*`comment`: ⟨dep-gen-event-stmt⟩ */
send:
  Piggyback : checkpoint vector, pred matrix
/* `comment`: ⟨ckpt-stmt⟩ */
Checkpoint : independent, asynchronous on local disk
Implementation : incremental
Scheduling policy : random
On checkpoint : Update checkpoint vector, pred matrix

Figure 5.4: Specification for the CIC protocols used in the study.

Application	Protocols	Number of Checkpoints Local	Forced	Total	Ckpt. Size (MB)	Exec. Time (sec.)	Overhead
bt	BCS	22	35	57	57.8	6725	182.4%
	HMNR	22	35	57	57.9	6713	181.9%
	BQC	27	80	107	39.2	7320	207.4%
cg	BCS	7	16	23	12.0	1961	38.6%
	HMNR	7	16	23	11.7	1953	38.0%
	BQC	8	38	46	6.6	2195	55.1%
lu	BCS	6	13	19	10.9	1381	29.8%
	HMNR	6	13	19	11.0	1380	29.7%
	BQC	6	38	44	6.5	1601	50.5%
sp	BCS	6	13	19	21.7	1597	53.9%
	HMNR	6	13	19	21.8	1603	54.4%
	BQC	8	51	59	12.0	2050	97.5%
mg	BCS	12	21	33	54.8	3100	192.2%
	HMNR	12	19	31	59.9	3020	184.6%
	BQC	14	53	67	30.3	3550	234.6%

Table 5.1: Performance of three CIC protocols using blocking checkpointing with $\mu = 360$ sec.

5.3.2 The Measured Performance of CIC Protocols with Blocking Checkpointing

Table 5.1 shows the results of the experiments for all the applications under the three protocols. Local checkpoints are triggered according to an exponential distribution that has a mean checkpoint interval (μ) set to 360 seconds. We report the execution time of the entire application, the per-process average number of local and forced checkpoints, and the per-process average checkpoint size (either local or forced). For convenience, the average per-process total number of checkpoints is also reported.

The running time overhead is high—between 30% and 235% of the execution time. Such a high overhead is actually common in systems where checkpoints are not coordinated and processes communicate periodically in "rounds" [24]. Whenever a process takes a synchronous checkpoint, it inevitably slows down because of the state saving and memory copying that occur during the checkpoint. This in turn delays the production of the results that the process is to send to others in the next communication round. Consequently, the slowdown affects other processes even if they are not taking checkpoints. Table 5.1 shows that this effect has a much more adverse impact on bt and mg applications because the checkpoint sizes in bt and mg are large (about 50MB in BCS, HMNR and about 30MB in BQC).

For the remaining experiments in this chapter, as we discussed in Section 5.2, CIC protocols are used with non-blocking checkpointing.

5.3.3 The Measured Performance of CIC Protocols with Non-blocking Checkpointing

Table 5.2 shows the results of our experiments with $\mu = 360, 480$. The table has a structure similar to that of Table 5.1. The results show that the performance overhead using non-blocking checkpointing is between 1% and 25% of the execution time. This overhead is due to the computation required to process the piggybacked information and the cost of forking child processes which save checkpoints. However, when compared to Table 5.1, the performance overhead is substantially less.

The results reveal a few issues. In BCS and HMNR, the number of forced checkpoints is essentially the same. In contrast, the BQC protocol incurs a comparatively larger number of forced checkpoints when compared to the other two. The reason for this can be ascribed to the communication pattern of the applications under study. These applications use a common iterative structure to solve a computationally intensive problem in which processes exchange partial results and resume. This leads to a communication pattern that mimics a periodic broadcast. Under this pattern, the BQC protocol seems to be too "eager" in preventing Z-cycles compared to BCS and HMNR. Furthermore, two additional effects seem to occur:

1. Many suspect Z-cycles end up causing forced checkpoints without actually being a

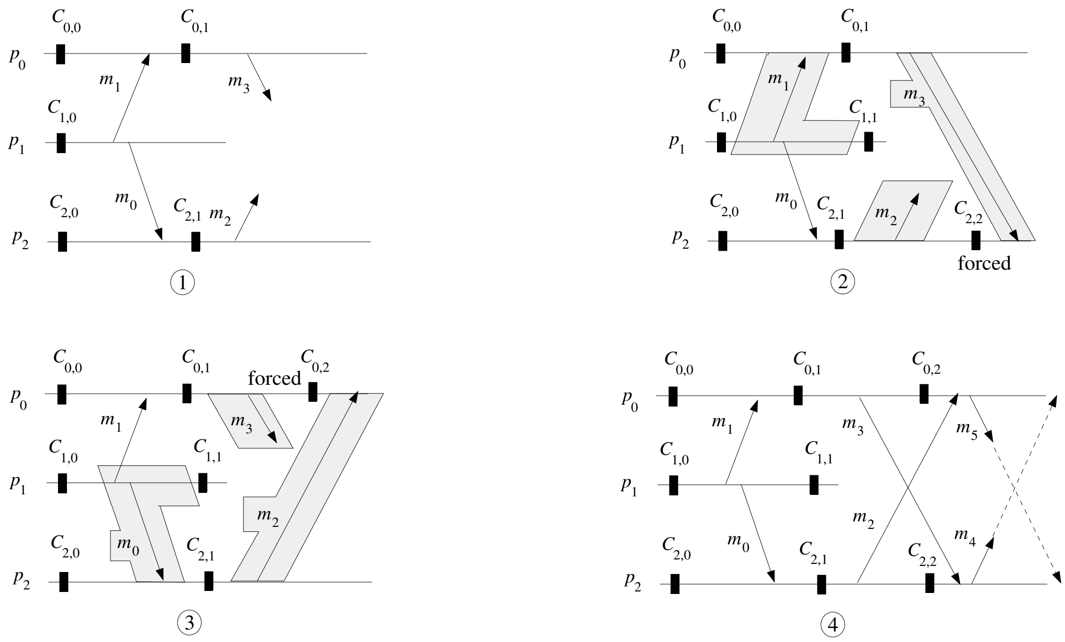


Figure 5.5: Anomalies in detecting suspect Z-cycles.

menace.

2. It is often the case that more than one process "volunteer" in parallel to break the same suspect Z-cycle by forcing checkpoints.

Consider the example shown in Figure 5.5. Process p_2 takes a forced checkpoint because of message m_3, not knowing that process p_1 has already broken the suspect Z-cycle (part (2) of the figure). Similarly, process p_0 takes a forced checkpoint because of message m_2, not knowing that process p_1 has already broken the Z-cycle using checkpoint $C_{1,1}$. This behavior continues for a while under the communication pattern used by our applications. This suggests that there is a disadvantage to using CIC protocols that suspect Z-cycles or that are eager to prevent a Z-cycle from forming before it is actually clear that one is indeed forming. In contrast, protocols that use time-stamping functions adopt a lazy approach: they prevent the formation of Z-cycles only at the last possible moment, and therefore perform better.

Application	Protocols	μ (sec.)	Number of Checkpoints Local	Forced	Total	Ckpt. Size (MB)	Exec. Time (sec.)	Overhead
bt	BCS	360	9	21	30	68.8	2555	7.3%
		480	6	14	20	71.2	2452	3.0%
	HMNR	360	9	20	29	70.5	2513	5.6%
		480	6	13	19	71.3	2430	2.0%
	BQC	360	11	70	81	40.0	2970	24.7%
		480	7	30	37	41.6	2750	15.5%
cg	BCS	360	5	10	15	16.9	1437	1.6%
		480	4	8	12	21.3	1429	1.0%
	HMNR	360	5	10	15	17.0	1438	1.6%
		480	4	8	12	21.1	1435	1.4%
	BQC	360	5	25	30	9.2	1506	6.4%
		480	4	16	20	12.8	1449	2.4%
lu	BCS	360	4	8	12	11.6	1133	6.5%
		480	3	5	8	11.9	1125	5.7%
	HMNR	360	4	8	12	11.6	1141	7.2%
		480	3	5	8	11.8	1133	6.5%
	BQC	360	4	30	34	6.5	1168	9.8%
		480	3	23	26	7.5	1152	8.3%
sp	BCS	360	4	8	12	22.6	1117	7.6%
		480	3	5	8	23.2	1103	6.3%
	HMNR	360	4	9	13	22.5	1117	7.6%
		480	3	5	8	23.3	1101	6.1%
	BQC	360	4	35	39	11.5	1174	13.1%
		480	3	23	26	13.5	1136	9.4%
mg	BCS	360	5	11	16	76.8	1230	15.9%
		480	4	6	10	77.2	1210	14.0%
	HMNR	360	5	11	16	74.5	1237	16.6%
		480	4	6	10	76.1	1206	13.7%
	BQC	360	6	35	41	32.9	1275	20.2%
		480	4	23	27	34.2	1255	18.3%

Table 5.2: Performance of three CIC protocols for two checkpoint intervals using non-blocking checkpointing.

The results also show that the alleged benefit of process autonomy in placing local checkpoints does not materialize in practice. Under the best circumstances, a process takes twice as many forced checkpoints as local ones. The curious notion of process autonomy in distributed systems where all processes become inter-dependent seems to be on shaky ground.

The results also point out to another serious problem with CIC protocols in general, namely the unpredictability of the checkpointing rate. In all experiments, the protocols ended up taking more checkpoints than could be anticipated based on the local distribution of checkpoint placement. For BCS and HMNR, the number of forced checkpoints was generally twice the number of local ones. For BQC, the ratio was worse. The ratio in itself is a function of the application, the number of processes, and the checkpoint placement. The fact that it is unpredictable makes the protocols cumbersome to use in practice, because it is difficult to plan ahead of time the actual stable storage requirements and the mean checkpointing interval. Contrast this with consistent checkpointing protocols where the number of checkpoints and required stable storage can be estimated with great certainty beforehand [24].

In summary, lazy protocols that use time-stamping to break Z-cycles perform better than eager protocols that take forced checkpoints as soon as they suspect a Z-cycle. The unpredictability of the actual number of checkpoints to be taken (forced and local) makes these protocols cumbersome to use in practice because no reasonable planning of resources and checkpointing frequency can be made without understanding the application and its communication patterns. Also, it seems that any hope of a benefit for allowing processes to take independent checkpoints is thwarted by the fact that a process ends up taking at least twice as many forced checkpoints than local ones. Finally, CIC protocols share some of the negative performance properties of independent checkpointing when used in computations where the processes are tightly coupled and communicate frequently.

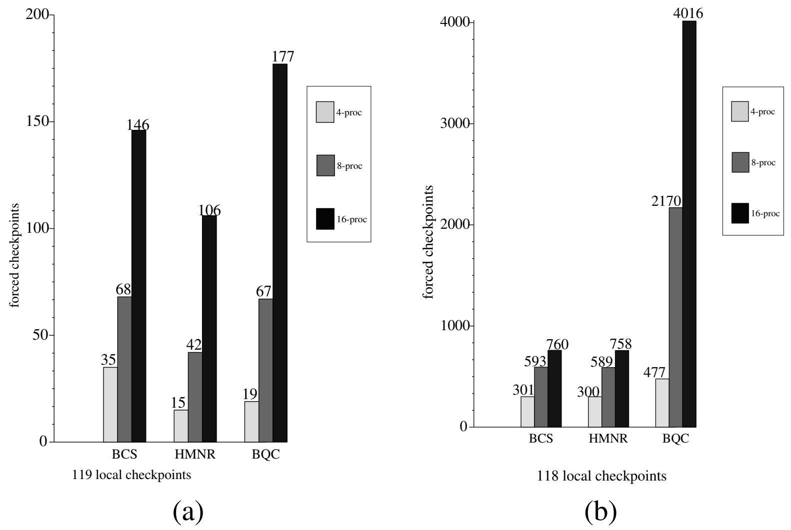


Figure 5.6: The effect of adaptive local checkpointing: Number of local and forced checkpoints for the three protocols under values of T for the four applications under study.

5.3.4 Scalability and Effects of Communication Patterns

To assess the effect of increasing the number of processes on the protocol performance, we constructed a simulator to measure the number of forced checkpoints for each of the three protocols. We first validated the simulator using the measured number of forced checkpoints for 4 processes, and we then used it to estimate the number of forced checkpoints for different numbers of processors and different communication patterns. Figure 5.6 shows the results for the three protocols as the number of processes in the computation varies. Two different sets of measurements are reported. The first set is for a random distribution of messages with a relatively low load, where each process sends an average of 10 messages between each two consecutive local checkpoints. That is, in this simulation processes do not communicate much and communicate with different processes equally at random and at random intervals. During this simulation, 119 local checkpoints were taken on average. The second set of measurements shows the same results but with a different communication pattern in which each process talks to two designated neighbors at uniform intervals.

A process sends about 500 messages between each two consecutive local checkpoints. This communication pattern is representative of those that occur in distributed over-relaxation algorithms.

The results show that, in general, the communication pattern strongly affects the behavior of CIC protocols. This is expected. But the results also show that CIC protocols do not scale well. In both cases, there is an almost linear increase in the number of forced checkpoints per process as the number of processes increase. For this set of applications, at least, the conventional wisdom that these protocols scale better because they do not resort to global coordination is not true. The results also show that CIC protocols seem to favor random patterns of communications with low loads.

5.3.5 An Adaptive Local Checkpointing Policy

The results of the experiments so far suggest that a flurry of forced checkpoints occur throughout the system as a result of one process taking a local checkpoint. It is plausible that if forced checkpoints are not taken into account, a local checkpointing policy may take a local checkpoint shortly after a forced checkpoint has been taken. Such a local checkpoint advances the recovery point of the process by a very short amount compared to the previous forced checkpoint. Furthermore, this local checkpoint is likely to trigger more forced checkpoints in other processes, escalating the phenomenon even further. It may be argued that the resulting overhead can be limited by using incremental checkpointing, and that therefore the local checkpoint does not have to save a lot of state on stable storage if a forced checkpoint has been taken recently. But we contend that taking a checkpoint, however small, always has an overhead associated with it, if only to compute the state that must be saved and to arrange for the copy-on-write to implement non-blocking checkpointing. However it may be, this overhead cannot be ignored. Therefore, there is very little to gain by taking this local checkpoint, while there is a potential for larger overhead. To fix this problem, we experimented with an adaptive local checkpointing policy that refrains from taking a scheduled local checkpoint if a forced checkpoint has occurred during the last T seconds, where T is a parameter.

Figure 5.7 shows the resulting number of local and forced checkpoints for the four applications and the three protocols under study. We report three measurements, one with the adaptive policy disabled ($T = 0$) and two for different values of T (60 and 90 seconds). For each T, the figure shows the number of local and forced checkpoints under each of the three protocols. The measurements for different applications are reported separately.

The results show that taking forced checkpointing into account reduces the number of local checkpoints, which in turn reduces the number of forced checkpoints. The results are more pronounced for the BQC protocol. These results demonstrate two effects:

1. A successful local checkpoint placement policy must contain a dynamic element that accounts for the occurrence of forced checkpoints. The simple policy "checkpoint every x seconds" does not work well.

2. A successful local checkpoint placement policy must adapt to the application communication patterns as they change during execution. This would reduce the frequency of checkpoints when the communication load is heavy and the frequency of forced checkpoints is high, and vice versa.

5.4 Summary

We performed several experiments to analyze the behavior and characteristics of communication-induced checkpointing for a class of compute intensive distributed applications. Our results show that:

1. CIC protocols that use an eager approach to preventing Z-cycles by taking forced checkpoints whenever they suspect the formation of a Z-cycle are bound to perform worse than lazy protocols that use a time-stamping function to prevent a Z-cycle at the last possible second.

2. CIC protocols do not scale well with a larger number of processes. We have found that the number of forced checkpoints increases almost linearly with the number of processes.

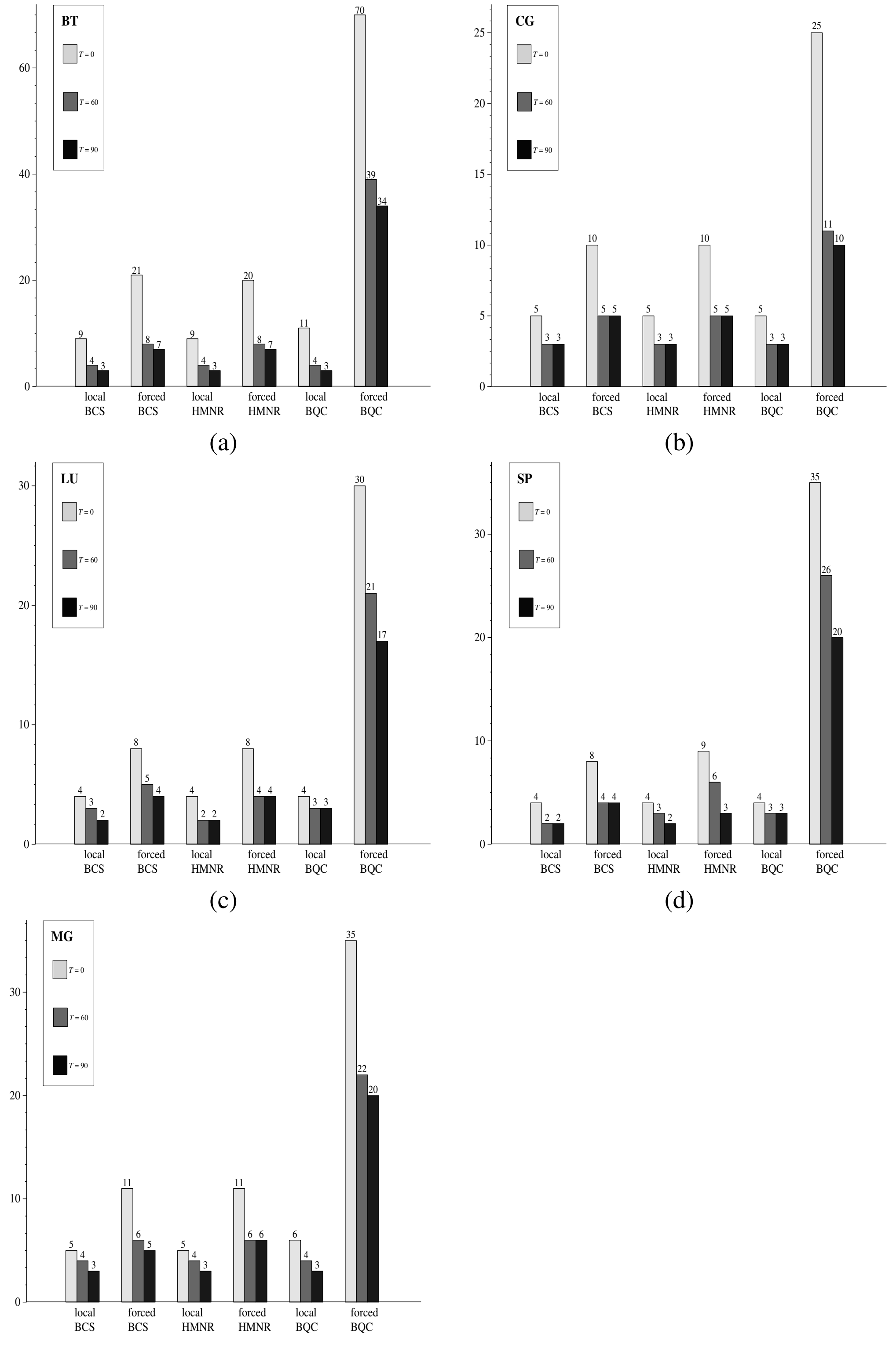
3. A process takes at least twice as many forced checkpoints as local ones. Therefore, the touted benefit of autonomy of CIC protocols in allowing the processes to take independent checkpoints does not seem to materialize in practice.

4. There is a considerable unpredictability in the way CIC protocols behave in practice. The amount of stable storage required, performance overhead, and number of forced checkpoints depend greatly on the number of processes, the application, and the communication pattern. This unpredictability makes the use of CIC protocols in practice more cumbersome than other alternatives.

5. A successful placement policy of local checkpoints must be dynamic, must account for forced checkpoints, and must adapt to changes in the application behavior.

6. CIC protocols seem to perform best when the communication load is low and the pattern is random. Regular, heavy load communication patterns seem to fare worse.

These results, while valid only for the application set that we have studied, do provide sufficient evidence to suspect that much of the conventional wisdom about these protocols is questionable.



(a) (b) (c) (d) (e)

Figure 5.7: The effect of adaptive local checkpointing: Number of local and forced checkpoints for the three protocols under different values of T for the five applications under study.

Chapter 6

Evaluating Process Interaction With the Environment

Rollback-recovery protocols such as checkpointing and optimistic and causal message-logging impose a low-overhead on the performance of applications in which processes communicate only by exchanging messages [26]. However, it is generally believed that rollback-recovery protocols impose a high overhead when used by applications in which processes also interact with the external environment. Prior to sending an output to the environment, rollback-recovery protocols require a process to execute an output-commit protocol; this protocol saves to stable storage synchronously information to guarantee that the state in which the output is generated is recoverable. Rollback-recovery protocols employ output-commits because they assume that the environment cannot be trusted to support crash recovery.

Prior research in rollback recovery has focussed on reducing the overhead of output-commit. Compared to checkpointing protocols, message-logging protocols support fast output-commits: In checkpointing protocols, an output-commit requires processes to take a coordinated checkpoint; whereas, in message-logging protocols, an output-commit can be accomplished by saving on stable storage the determinants of non-deterministic events that causally precede the event in which output is generated to the environment. Between low-overhead optimistic and causal protocols, causal protocols provide better support for fast output-commits: Causal protocols [5, 25] allow processes to output-commit independently, whereas optimistic protocols [20, 78] require co-ordination amongst processes for an output-commit.

Although qualitatively small, causal protocols could incur significant overhead for output-commits. To illustrate, consider applications in which processes communicate by exchanging messages as well as reading and writing shared files. Even though rollback-recovery protocols rely on file servers to provide stable storage for checkpoints and other information used during recovery, the same protocols, however, treat file servers as generic components of the external environment whenever processes communicate by reading or writing other files. We illustrate the need for synchronous logging during file sharing using the example shown in Figure 6.1. Processes p, q, and r exchange messages and share a file F. Process p reads from F, sends message m_0 to q, and then fails. Process q delivers m_0, writes to F, sends message m_1 to r, and then fails. Process r eventually delivers m_1. Let $F.v_0$ be the version of F accessed by p and let $F.v_1$ be the new version of F created by q. During recovery, p will again read file F. However this time, instead of $F.v_0$, the file server will provide $F.v_1$ to p. Because p reads a different version of F during recovery, p may not re-send m_0, or indeed any message, to q, leaving p and q in mutually-inconsistent states. To avoid such inconsistencies, rollback-recovery protocols proceed as follows: when a process reads or writes a file, it blocks until the information necessary to prevent inconsistencies during recovery is logged to stable storage, which is typically implemented using a disk subsystem. Prior to a write, a process executes an output-commit; on a read, a process saves on disk the content of the file read, to guarantee that it will be available during recovery.

Although output-commits may impose a high overhead on application performance, the significance of the overhead has not been studied. In the first part of this chapter, we use Egida to quantify the overhead incurred by processes whenever they interact with the file

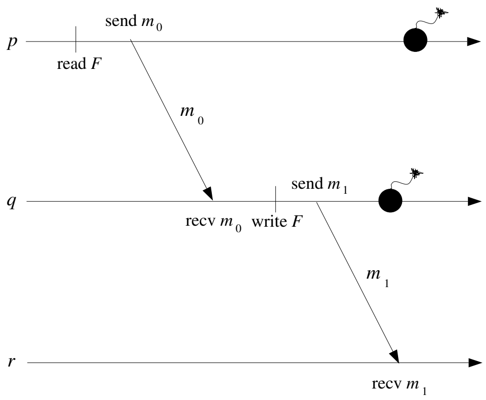


Figure 6.1: Example demonstrating the need for synchronous logging resulting from file sharing.

server. Our results show that the resulting overhead is significant and it is likely to increase as the scale of the applications and the disparity between processor and disk speeds continue to increase.

To reduce the logging overhead incurred by processes whenever they interact with the external environment, two approaches have been proposed:

- Efficient implementation of stable storage with a weaker failure model. For example, the Rio [19] file cache provides persistent storage using volatile memory and it can be used to store the output-commit information with a low-overhead on performance [51]; however, Rio is vulnerable to hardware failures.

- Batch the output-commits [81] to improve output-commit performance and delay releasing the output to the environment.

In this chapter, we pursue a different approach—treat parts of the environment as integral components that can be trusted to support failure recovery. As a result, message-logging protocols can perform dependency-tracking whenever a process interact with the environment and thereby avoid an output-commit. We instantiate this approach by developing a novel file-sharing protocol that reduces the logging overhead incurred by processes when they interact with the file server. Specifically, our solution is to track causal dependencies resulting from file sharing and to record them using determinants. To ensure determinants availability, we extend causal logging protocols [5] to also replicate the determinants created during file sharing. The availablility of determinants during recovery allows a process' interactions with the file server to be reproduced, and file data lost in a failure to be regenerated. A side-effect of our file-sharing protocol is that it allows us to introduce a novel concept—implementation in volatile memory of stable storage for files. Traditionally, a file F modified by an process p can be shared by another process q only after p has synchronously written F to disks at the file server [7, 33]. In our solution, no synchronous write is needed, and p can send F to q without delays. As a consequence, our solution reduces the fault-tolerance overhead for applications that communicate through both message passing and file sharing to that incurred by conventional low-overhead fault-tolerance protocols for purely message passing applications.

Although presented for the file system, the concept of trusting portions of the environment to support application's recovery is more general; yet it is not universal. For instance, a database is an entity of the environment that can be rolled back to support application's failure recovery. On the other hand, instead of a database, suppose that a process were to send an output to a printer. Once the output is printed, the state of the printer cannot be rolled back. Therefore, in this case, it is necessary to execute an output-commit before releasing output to the environment.

The remainder of the chapter is organized as follows. The overhead incurred by processes when they interact with the environment, i.e., the file server, is quantified in Section 6.1. We describe our file-sharing protocol and its salient features in Sections 6.2 and 6.3, respectively, and then discuss the protocol implementation issues in Section 6.4. Section 6.5 discusses related work and finally, Section 6.6 summarizes our results.

6.1 Experimental Evaluation of Process Interaction with Environment

6.1.1 Experimental Setting

To quantify the overhead incurred by processes whenever they interact with the file server, we consider two environments `FP` and `FF`:

- `FP` is an environment in which processes may be prone to failures. To recover from a crash, processes use causal protocols [5]. During file sharing, causal protocols require a process (1) to execute an output-commit before the process writes to a file and (2) to log synchronously on stable storage the content of a file read by the process. Also, in `FP`, the file server performs a synchronous writeback before a modified file can be shared amongst processes. This is because existing file systems [7, 33] assume that processes cannot be trusted to regenerate data lost in a crash.

- `FF` is a failure-free environment. In this case, since failures cannot occur, there is no need for processes to use any rollback recovery protocol. Consequently, during file sharing, a process neither executes an output-commit nor logs the content of a file it reads. Also, in `FF`, the file server allows modified files to be shared by processes without any synchronous writeback.

In this study, we use Egida for both `FP` and `FF`. Figure 6.2 shows the specifications for the protocols used in `FP` and `FF`. Note that in `FF` no logging is performed for a read, write, send, or receive events.

6.1.2 Implementing Consistency for Shared Files

The file server used in our performance studies implements the NFS protocol [1]. This protocol does not support the single-writer, multiple-reader file sharing policy defined by our system model (see Section 2.1). Therefore, for this performance study, we emulate the file

/* `comment:` ⟨non-det-event-stmt⟩ */
receive:
  determinant : {source, sesn, dest, desn}
  Log determinant on volatile memory of self,
    NFS disk
  Flush : NULL
  Log message on volatile memory of self

read :
  determinant : {desn}
  Log determinant on NFS disk
  Log data on NFS disk

/*`comment:` ⟨dep-gen-event-stmt⟩ */
send:
  Piggyback : determinants
  Log message on
    volatile memory of self

/* `comment:` ⟨output-commit-stmt⟩ */
write:
  Output commit : determinants
  Protocol : independent

/* `comment:` ⟨ckpt-stmt⟩ */
Checkpoint : independent, asynchronous
  on local disk

/* `comment:` ⟨recovery-stmt⟩ */
Recovery :
  Protocol to retrieve determinants : distributed
  Protocol to detect orphans : NULL

(a)

/* `comment:` ⟨non-det-event-stmt⟩ */
receive:
  determinant : NULL
  Log determinant on NULL

read:
  determinant : NULL
  Log determinant on NULL

/*`comment:` ⟨dep-gen-event-stmt⟩ */
send:
  Piggyback NULL
  Log message on NULL

(b)

Figure 6.2: Specification of (a) causal message logging protocol used in `FP` and (b) the protocol for `FF` in which data is logged.

server using a process that does not participate in the computation. The emulated file server is used for storing only those files accessed by the application. In our experiments, whenever a process executes a read event, in both FP and FF, the corresponding event handler in Egida acquires read-ownership from the emulated file server and then performs the read; write events are handled similarly. Also, for our experiments, whenever a process acquires file ownership (for reading/writing), Egida caches the file until the ownership is revoked by the emulated file server. Hence, as long as a process has the appropriate file ownership, Egida's read/write event handlers use the cached copy to execute the corresponding read/write event.

6.1.3 Methodology

We model applications using a synthetic program that allows us to evaluate different patterns of process interaction with the environment. In our model, an application consists of N processes that communicate by exchanging messages and by reading/writing files. Each of the N processes each executes a sequence of send, deliver, read, write, and other local events. The relative frequencies of these events is governed by P_c, P_m, and P_f, which, respectively, denote the fractions of local, message send/deliver, and file read/write events executed by a process. Also, the extent of file sharing is controlled by P_s, which denotes the probability that the file accessed by a process is being shared concurrently with other processes.

6.1.4 Results

We measure the execution time of processes in FF (denoted by E_{base}) and the increase in the execution time yielded by synchronous write-back, read logging, and output commit in FP (denoted by I_s, I_r, and I_o, respectively). Figure 6.3 plots how $O_s = I_s/E_{base}$, $O_r = I_r/E_{base}$, and $O_o = I_o/E_{base}$ vary with system parameters (namely, N, P_c, P_m, P_f, and P_s). The results are shown with 95% confidence intervals.

- Figures 6.3(a) and 6.3(b) demonstrate that increasing N and P_s increases the possibility that, when an process executes a read or a write event, the most recent version of the file is with a different process. Consequently, I_s and O_s increase. Since changes in N or P_s do not alter the size of files read or the number of messages delivered by processes, I_r and I_o do not change with N and P_s. However, since E_{base} increases with N and P_s, both O_r and O_o decrease marginally.

- Increasing the ratio P_f/P_m increases the frequency of file read and write events with respect to send and deliver events. This has two effects (see Figure 6.3(c)):

  1. It increases the frequency of synchronous write-backs and read logging. Hence, O_s and O_r increase with P_f/P_m.
  2. It increases O_o until $P_f/P_m \leq 1$. When $P_f/P_m \leq 1$, successive write events executed by a process are separated by one or more message send or deliver events. Hence, increasing P_f/P_m increases the frequency of output commits, and the value of O_o. Once $P_f/P_m > 1$, a process may not execute any send or deliver events between successive write events. Consequently, although the frequency of write events increases with increase in P_f/P_m, the number of invocations of the output commit protocol, and hence O_o, decreases.

- Figure 6.3(d) demonstrates that increasing $(P_m + P_f)$ makes the application both communication-bound and I/O-bound, increasing O_s, O_r, and O_o.

In the results we have presented so far, the execution times of events depend on parameters such as the processor speed as well as the latency and the bandwidth of disks and networks. Traditionally, technological advances have resulted in improvements in the specific values of these parameters and trends [7] indicate that further improvements are likely (see Figure 6.4). To account for these trends, we developed a simulator and performed a set of trace-driven simulations. We constructed the simulator such that (1) it models the same synthetic application that we have used in the experiments so far and (2) it executes instruction traces that are obtained by running the synthetic application. We use the simulator to perform simulations in which we account for the expected technological improvements in processor speed as well as the latency and the bandwidth of disks and networks. Figure 6.4 shows the results of the simulations which indicate that $\mathcal{O}$ will more than triple over the next few years.

In these experiments, we assume that `FP` blocks a process executing a read or a write event until the completion of the read logging or output commit protocols. To guarantee consistent recovery, however, it suffices that the result of the read is logged on disk prior to executing subsequent message sends or file writes. Similarly, unless file sharing or some other application requirements force the write to the file server to be synchronous, most files servers batch multiple write events to achieve better performance. In this case, it may be possible to execute the output commit protocol asynchronously, albeit it must complete prior to the corresponding batched write [40]. Figure 6.4 demonstrate that the above optimizations do reduce $\mathcal{O}$, but due to the orders of magnitude difference between memory and disk speeds, synchronous writes to the disk significantly degrade application performance.

From these experiments, we conclude that rollback-recovery procotocols cause applications incur a substantial overhead whenever processes interact with the environment. We believe that the main cause of this overhead is the *mutual mistrust* between these fault-tolerance protocols and file servers: the protocols do not trust file servers to provide during the recovery of a failed process the files read by that process prior to failure; and the file servers do not rely on the protocols' ability to recover failed processes to improve application performance. In what follows, we discuss the ideas that underlie our approach to eliminating this mistrust.

6.2 Low-Overhead Protocol For Fault-tolerant File Sharing

In this section, we propose a novel approach that, through mutual cooperation between application-level failure recovery protocols and file servers, eliminates the fault-tolerance

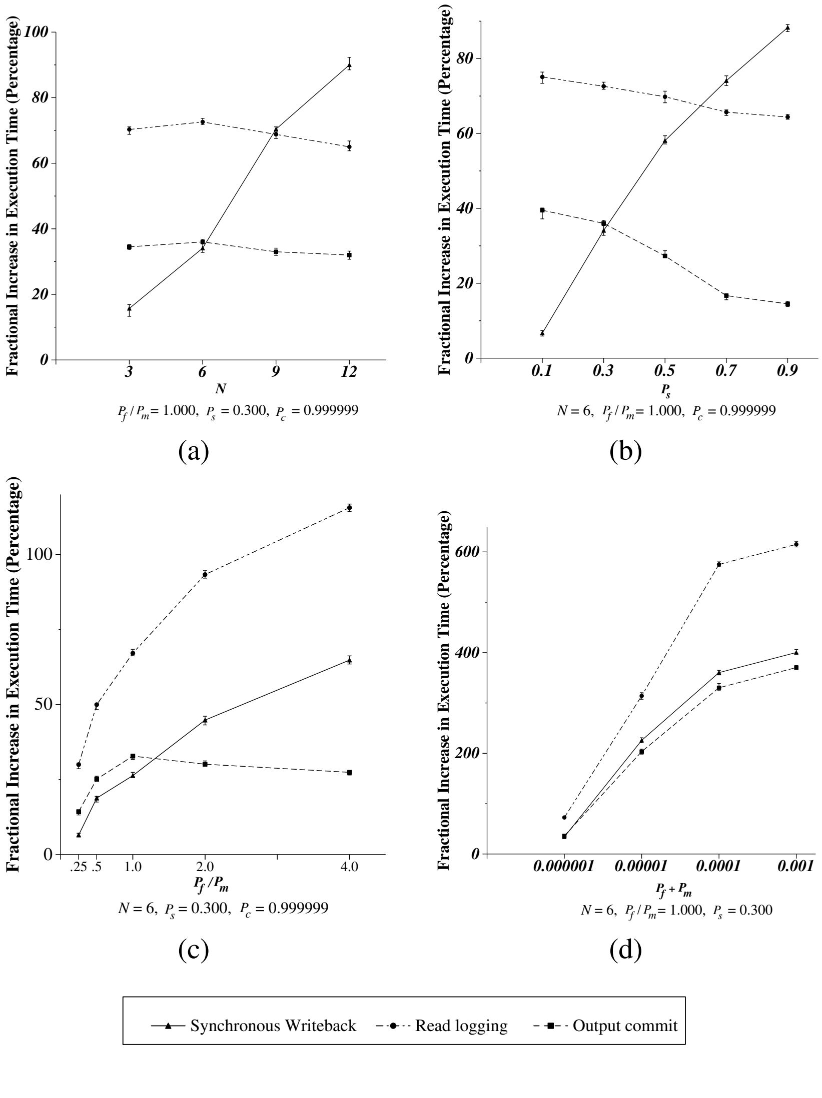


Figure 6.3: Variation in O_s, O_r, and O_o with N, P_c, P_f, P_m, and P_s.

Parameter	1997	Improvement Rate
Processor Speed	200 MIPS	50%/year
Network Latency	1ms	20%/year
Network Bandwidth	20 MB/s	45% /year
Disk Latency	12ms	10%/year
Disk Bandwidth	6 MB/s	20%/year

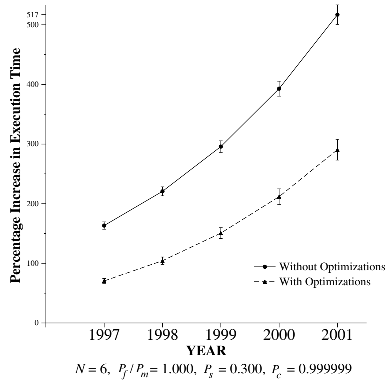


Figure 6.4: Effect of technological trends and optimizations on $\mathcal{O}$

overhead incurred by applications in which processes communicate through file sharing. Our approach has two advantages:

1. It frees processes from the twin obligations of synchronously logging all the data they read and performing synchronous output commits whenever they write.

2. It allows recovering processes to regenerate files that were lost when these processes crashed. Thus, the file server can allow modified files to be shared among processes without first requiring that these files be synchronously written to disks.

To attain these advantages, for each non-deterministic event (*i.e.,* each deliver, read, or write event) we need (1) to identify the information necessary to reproduce that event during recovery, (2) to design efficient protocols that guarantee that this information will be available during recovery, and finally (3) to specify how this information is used during recovery. We address the first two points in the remainder of this section, and then discuss recovery and implementation issues in Section 6.4.

6.2.1 Reproducing Non-Deterministic Events

We assign to each non-deterministic event e a unique determinant, which we denote by $\#e$. The purpose of $\#e$ is to contain the information necessary to reproduce e during recovery.

A determinant has three components: (1) the process executing the non-deterministic event, (2) the relative order of this event with respect to other non-deterministic events executed by the same process, and (3) the data that is returned by executing the event. Depending on the event, this data can be either the content of a message, or the content of a file that is either read or written.

To record the relative order in which events are executed, each process increments a local counter for every send, deliver, read, or write event. We call this counter *event sequence number*, or esn. Then, for an event e that returns some data $data$, the determinant can be expressed by the tuple $\langle dest, desn, data \rangle$, where $dest$ and $desn$ are, respectively, the process executing e and the esn assigned to e by that process.

To be practical, determinants must be small, so that they won't be costly to replicate or write to disk for fault-tolerance. We can significantly reduce the size of determinants by observing that they do not need to include $data$ explicitly: instead, it suffices that determinants contain some way of uniquely identifying $data$ during recovery. In fact, if there is a unique way to associate $data$ with the corresponding determinant, and all determinants are available during recovery, then a simple induction on the length of the execution shows that delivered messages and file versions accessed for reading or writing can be deterministically regenerated [4]. Therefore, assuming that we can guarantee the availability of determinants (see Section 6.2.2), we substitute $data$ in $\#e$ with following pair: $\langle source, sesn \rangle$. The meaning of $source$ and $sesn$ depends on e. If e is a deliver event for a message m, then m is uniquely identified by storing in $source$ the identity of the sender of m and in $sesn$ the esn assigned by $source$ to the corresponding send event. If instead e is a read or a write event for a version v of file F, then $F.v$ is uniquely identified by storing in $source$ the identity of the creator of $F.v$ and in $sesn$ the esn assigned by $source$ to the write event that created $F.v$. Thus, in its final form a determinant has the following structure: $\#e = \langle dest, desn, source, sesn \rangle$.

Hence, the determinants are small in size and have the same structure independent of the type of non-deterministic event they refer to.

6.2.2 Logging Determinants Efficiently

For logging determinants, we use Family-based Logging (FBL) [2], an instance of causal logging protocols. FBL protocols was originally designed to log determinants for deliver events. However, because the determinants of read, write, and deliver events that we have derived have an identical structure, we can naturally extend FBL protocols to handle communication resulting from file sharing. The result is a simple, uniform approach, which can provide low-overhead fault-tolerance to applications in which communication is performed through message passing, file sharing, or a combination of the two. In what follows, we briefly describe the main features of FBL [2] and discuss how we generalize FBL to handle read and write determinants.

6.2.2.1 Family-Based Logging

FBL protocols [2] are based on the following observation: in a system where processes fail independently and no more than f processes fail concurrently, stable storage can be implemented by replicating determinants in the volatile memory of $f + 1$ processes [3]. FBL's logging technique can be specified using the following two sets for each non-deterministic event e [2]:

Depend(e) This set includes the process which executed e and any process that executed an event e' such that $e \to e'$.

Log(e) This set includes the processes that maintain a copy of $\#e$ in volatile memory.

FBL guarantees that the following property holds [2]:

$$\Box(\forall e : \neg stable(e) : (Depend(e) \subseteq Log(e)) \wedge \Diamond(Depend(e) = Log(e))) \qquad (6.1)$$

here $\Box$ and $\Diamond$ are, respectively, the "always" and "eventually" temporal operators [62]. This property strongly couples logging in FBL with tracking of causal dependencies involving non-deterministic events. It guarantees that if the state of a correct process p causally depends on an event e, then either e is stable (*i.e.* either $|Log(e)| > f$ or $\#e$ has been written to disk), or p has logged a copy of $\#e$ in its volatile memory. Furthermore, if p has logged a copy of $\#e$, then p will eventually depend on e. Note that this scheme ensures that even if a failure occurs before $\#e$ is replicated $f + 1$ times, if p causally depends on e, then $\#e$ is not lost.

6.2.2.2 Family-Based Logging for Efficient File Sharing

A process p becomes a member of $Depend(e)$ either by delivering a message or by reading or writing a file. In either case, we need to guarantee that p first becomes a member of $Log(e)$. In message passing applications this is accomplished by piggybacking determinants on existing application messages. We now show how to apply a similar scheme to the dependencies generated by file sharing.

Current file servers provide no mechanisms to enforce Property 6.1. In fact, it is precisely because of the file server's inability to enforce this property that processes perform a synchronous output commit protocol before writing a file. A simple way to address the file server's lack of cooperation is to treat the file server as just another application process. While in FBL processes piggyback determinants only on messages, now processes could piggyback determinants also on files that are written back at the file server. The file server could keep files and determinants in its volatile memory and piggyback non-stable determinants to the files that it forwards to application processes. We improve on this scheme by observing that to satisfy Property 6.1 it is not necessary for determinants (or indeed even for files!—see Section 6.4.4) to be logged at the file server. It suffices that when a process obtains a file, it receives with the file the non-stable determinants logged in the volatile memory of the file's last writer. Figure 6.5 illustrates a file-sharing protocol based on this observation. After receiving a request to access file F from p, the file server forwards the request to the process that has the latest version of F (q in the figure); q then sends F to

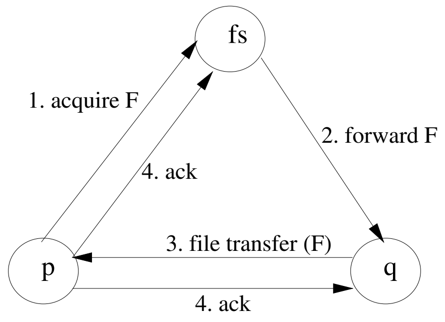


Figure 6.5: Example illustrating the file-sharing protocol

p, piggybacking on F all non-stable determinants in its volatile memory. The file transfer occurs directly between the processes, bypassing the file server that is used solely to locate the process with the latest version of F. For the purpose of the logging protocol, file sharing has become indistinguishable from message passing—the only difference is that to determine the destination of a "file-message", processes have to contact the file server. A formal description of our file sharing protocol is presented in Section 6.4.

6.2.3 Optimizations

Minimizing the number of determinants to piggyback is critical for the performance of FBL protocols. To address this problem, we refine protocol to reduce the number of determinants that are created in the first place. In particular, we observe that once a process has acquired ownership of a file version for reading or writing, then the outcomes of the subsequent read or write events executed by that process are uniquely determined for as long as the process owns the file. Therefore, it is not necessary to generate a new determinant for each read and write event: instead, it suffices to generate a determinant only when a process acquires ownership of a file. A naîve implementation of this optimization, however, may complicate recovery. Consider, for instance, an execution in which a process p executes several read events, and in doing so reads from two versions of the same file. If p fails, the recovery protocol must recognize which of the original read events applied to the first version, and which applied the second version. We solve this problem (1) by requiring that p increment its esn counter every time it executes a read, even if the read event does not result in the acquisition of a new version of the file; and (2) by generating a determinant only when p acquires a new file version. We treat write events similarly. The resulting determinants can be efficiently encoded using compression techniques discussed elsewhere [3].

We can reduce the number of file versions logged in the volatile memory of processes. Suppose a process p acquires write ownership on a file F and repeatedly modifies it before its ownership is revoked. Since all but the last version of F created by p are invisible to other processes in the system, p keeps only the last version of F in its volatile log.

6.3 Salient Features and Contributions

The proposed protocol has the following advantages:

- It eliminates synchronous output commits on writes. The protocol ensures that when a process executes an event e which results in the acquisition of a file, it receives with the file the non-stable determinant of any event that happened before e.

- It eliminates synchronous logging on reads, since files accessed during an execution can be regenerated during recovery.

- It eliminates synchronous write-backs required by the file server during file sharing. Processes can exchange files directly, without first saving the files on disks at the server. We make two observations about this implementation of stable storage.

  First, the protocol implements a notion of stability that is weaker than the one offered by existing file servers. In the latter, all data read by a process is stable. Our protocol only recovers the data necessary to restore the application to a consistent global state. For instance, if process p creates $F.v$, sends $F.v$ to q, and both p and q crash before communicating with other processes, our protocol does not guarantee that $F.v$ will be regenerated during recovery, since the state of no correct process causally depends on restoring $F.v$. We feel that this weaker notion of stability is closer to the real needs of the applications.

Second, high performance in our protocol does come at the cost of reduced availability (lost files may need to be regenerated). However, it is straightforward to change the protocol so that modified files are *asynchronously* written back to the file server, thereby making them readily available during recovery. The file server in this case needs to support file versioning: this is achieved, for instance, using a log-structured file system [68].

6.4 Protocol Design

In this section, we identify the issues we faced in designing our file-sharing protocol and present our solutions. We also provide the pseudo-code for handling file read/write events. Since the pseudo-code for causal logging is presented elsewhere [2], we do not include it in this section.

6.4.1 Notation

Event sequence number ESN_p is an integer, used to uniquely identify and order each send, deliver, read, and write event executed by p. ESN_p is initialized to 0.

We represent file versions and file-transfer messages as follows:

File versions We represent a version v of file F with the tuple $\langle fileId, version, data \rangle$ where $fileId$ is the unique identifier of F, $data$ is the content of $F.v$, and $version$ is a pair of the form $\langle creator, cesn \rangle$ in which $creator$ identifies the process that created $F.v$, and $cesn$ is the event sequence number of the write event that created $F.v$.

File-transfer messages We represent a message used to transfer $F.v$ from $source$ to $dest$ with the tuple $\langle source, dest, fileId, version, data, piggyback \rangle$, where $\langle fileId, version, data \rangle$

represent $F.v$, and $piggyback$ stores the determinants that are piggybacked on the file transfer message by $source$.

6.4.2 Logging Determinants and Files

Each process p maintains the following logs in its volatile memory:

Determinant Log : $DetLog_p$ contains one entry for each determinant $\#e$ logged by p. The entry dle in $DetLog_p$ corresponding to $\#e$ has the form $\langle source, sesn, dest, desn, logged\_at\rangle$, where the first four fields hold the same values of the corresponding fields in $\#e$ and $logged\_at$ is a set consisting of the processes p knows to have logged a copy of $\#e$. p considers $\#e$ stable when either $\#e$ has been written to disk or $|dle.logged\_at| > f$.

File Log : $FileLog_p$ contains an entry for each file version that p forwards to another process. Note that p forwards only those file versions that p created. The entry fle in $FileLog_p$ corresponding to file version $F.v$ has the form $\langle fileId, version, data, dv, destSet\rangle$ where the first three fields hold the same values of the corresponding fields in $F.v$, dv is a vector of N event seqeunce numbers, called *dependency vector* [2], and $destSet$ is a set consisting of the processes to whom p sent $F.v$. $dv[q]$ counts the number of send, deliver, read and write events executed by q that causally precede the write event that created $F.v$.

6.4.3 File Sharing

To achieve better performance, processes cache in their volatile memory those files for which they have acquired ownership. We say that a process p has a *valid* copy of a file in its cache until the file server revokes p's ownership for both reading and writing. To implement a single-writer, multiple-reader policy, the file server uses the following data structure:

File Table : $FileTable$ contains an entry for each file maintained by the file server. The entry fte for file F is of the form $\langle fileId, cv, versionSet, copySet, mode, pendingQ\rangle$

where $fte.cv$ identifies the latest version of F, $fte.versionSet$ contains an identifier for each previous version of F, $fte.copySet$ contains the identities of the processes with a *valid* copy of $F.cv$ in their volatile memory, $fte.mode$ records the mode (`READ` or `WRITE`) in which the processes in $fte.copySet$ have acquired $F.cv$, and $fte.pendingQ$ is a queue of pending ownership requests that have been received but not yet granted by the file server.

Before granting read ownership of F to p, the file server revokes the write ownership of the creator of $F.cv$. Before granting write ownership of F to p, the file server revokes the read ownership of the processes in $fte.copySet$ and both the read and the write ownership of $F.cv$'s creator. In either case, the creator forwards $F.data$ to p. When p receives $F.data$, p sends $F.version$ to the file server (Figure 6.5). The file server then sets $fte.cv$ to $F.version$. In addition, if p was granted read ownership, p is added to $fte.copySet$; otherwise, $fte.cv$ is added to $fte.versionSet$ and $fte.cv$ is set to $\langle p, \top \rangle$.

For caching files, each process p uses the following data structure:

File Cache : $FileCache_p$ contains an entry for each valid file version cached by p. The entry fce for file F is of the form $\langle fileId, version, data, mode, valid \rangle$ where $fce.mode$ records the mode (`READ` or `WRITE`) in which p has acquired ownership of $F.v$, and $fce.valid$ (`TRUE` or `FALSE`) indicates whether the fce is valid.

The pseudo-code for the read and write events executed by processes and for the file sharing protocol is shown in Figures 6.6- 6.13.

6.4.4 Failure Recovery

In causal logging protocols, crash recovery of process p proceeds in two phases. In the first phase, p retrieves from the logs of the remaining processes (1) the determinants and their corresponding result needed for replay and (2) the determinants p had logged prior to failure. In the second phase, p re-executes the original run using the determinants to replay correctly the result of non-deterministic events.

In our protocol, we use this same algorithm with the following modifications. In the first phase, we extend the set of determinants retrieved by p to include the determinants for file read and write events; also, p retrieves the content of the corresponding file versions. In the second phase, when p executes a file read/write event, that event may not have a corresponding determinant for replay. This may be because of the optimizations we introduced in Section 6.2.3 to suppress creating determinants during failure-free execution. To describe how those read/write events are handled, we first introduce the following notation:

$DetSet_q$ This set contains the determinants of events executed by p and logged by q. It includes all determinants $\#e$ with a corresponding entry dle in $DetLog_q$ such that $dle.dest = p$.

$FileSet_q$ This set contains the files sent by q to p.

$maxEsn_q$ This value is the event sequence number corresponding to the latest event of p that q depends upon. This value is used in Phase 2 to help p determine when recovery has completed.

During Phase 1, p collects $DetSet_q, FileSet_q$ and $maxEsn_q$ from each process q. At the end of Phase 1, p computes the following values:

$$DetSet_p = \cup_{q \in N} DetSet_q$$
$$FileSet_p = \cup_{q \in N} FileSet_q$$
$$maxEsn_p = \max\{maxEsn_q : q \in N\}$$

During Phase 2, when p executes a read/write event, the entry from $DetSet_p$ is matched with the corresponding entry from $FileSet_p$ using ESN_p; the entry from $FileSet_p$ is added to $FileCache_p$. However, if the read/write event has no corresponding determinant in $DetSet_p$, there are two cases to consider:

1. p's recovery may be complete—this occurs if ESN_p is equal to $maxEsn_p$;

2. otherwise, this event corresponds to the scenario during failure-free execution where p executed the event using the data in its cache. Hence, during recovery, p executes the same event using the latest version of the corresponding file in $FileCache_p$.

Finally, at the end of recovery, p uses the file server's $FileTable$ to determine which of the files in its cache are valid.

The pseudo-code corresponding to the read/write events executed by p during the roll-forward phase of recovery is shown in Figures 6.14 and 6.15.

6.5 Related Work

Our protocol builds upon the results of prior research on file systems and application-level fault tolerance protocols.

File servers that support read and write events efficiently have been a topic of much research. For instance, the log-structured file system employ techniques for improving the performance of file writes [68]; xFS implements cooperative caching to improve the performance of file reads [7]. Additionally, several techniques for designing file servers that assist applications during failure recovery have been investigated. For instance, versioning file systems eliminate the need for read logging by ensuring that a file version read by a process prior to failure will be available during recovery. Similarly, implementation of stable storage, using special hardware, such as non-volatile RAM [8], or a specialized operating system, such as the Rio file cache [19], in the memory sub-system at the server eliminates the need for synchronous writes to disks. Unfortunately, neither of these approaches address the problem of fault-tolerant file-sharing in its entirety. Versioning file systems do not eliminate the need for synchronous output commit and write-backs, and the implementations of stable storage in the server memory are vulnerable to hardware failures — a memory sub-system or processor failure can render the content of the stable storage inaccessible.

Our approach can tolerate concurrent crashes of multiple processes as well as hardware failures. It eliminates synchronous output commits on writes, synchronous logging on reads, and the synchronous write-backs required by the file server during file sharing.

6.6 Summary

In this chapter, we evaluated the performance of rollback-recovery protocols when used by applications in which processes also interact with the external environment. We focussed on one component of the environment, namely, the file system and evaluated the fault-tolerance overhead incurred by applications in which processes communicate both by exchanging messages and by reading and writing files. Our results showed that file sharing adversely affects the performance of a reliable application. The main cause of these adverse effects is the mutual mistrust between the file server and the protocols used to make applications fault-tolerant. In this chapter, we demonstrated that the artifacts of the mutual mistrust—namely, read logging, output commit, and synchronous write-backs to the file server—incur significant overhead, which, with the expected technological improvements, will more than triple over the next five years. We presented a novel approach that, through mutual cooperation between application-level failure recovery protocols and file servers, eliminates this overhead. Our approach (1) tracks causal dependencies resulting from file sharing using determinants, (2) efficiently replicates the determinants in the volatile memory of processes to ensure their availability during recovery, and (3) reproduces during recovery the interactions with the file server as well as the file data lost in a failure. Our approach allows processes to exchange files directly, without first saving the files on disks at the server. Thus, our solution virtually reduces the fault-tolerance overhead for applications that communicate through both message passing and file sharing to that incurred by conventional low-overhead fault-tolerance protocols for purely message passing applications.

$Read(fileId : \text{FileId}, data : \text{DataType}, result : \text{Boolean})$

```
if ∃fce ∈ FileCache_p st ((fce.fileId = fileId) ∧ (fce.valid = TRUE))
   result ← TRUE
   ESN_p ← ESN_p + 1
else
   send_acquire_ownership(filesystem, fileId, ⊤, READ)
   receive_data_source(filesystem, fileId, version, creator, result)
   /* comment : Read can fail if the file doesn't exist */
   if (result = FALSE)
      return
   receive_file_transfer(creator, fileId, version, data, piggyback)
   fce.fileId, fce.version, fce.data, fce.mode, fce.valid ← fileId, version, data, READ, TRUE
   FileCache_p ← FileCache_p ∪ {fce}
   for all e ∈ piggyback
      DetLog_p ← DetLog_p ∪ {⟨e, {creator, p}⟩}
   send_acknowledgement(creator, version.esn)
   send_transfer_complete(filesystem, fileId, version)
   ESN_p ← ESN_p + 1
   DetLog_p ← DetLog_p ∪ {⟨creator, version.esn, p, ESN_p, {p}⟩}
data ← fce.data
```

Figure 6.6: Process p executes $Read(fileId, data, result)$ during failure-free execution.

$Write(fileId : \text{FileId}, data : \text{DataType})$

if $\exists fce \in FileCache_p$ **st** $((fce.fileId = fileId) \wedge (fce.valid = \texttt{TRUE}) \wedge (fce.mode = \texttt{WRITE}))$
    $ESN_p \leftarrow ESN_p + 1$
    $fce.version.esn \leftarrow ESN_p$
else
    send_acquire_ownership $(filesystem, fileId, \top, \texttt{WRITE})$
    /* `comment`**:** For a write, the result is always TRUE */
    receive_data_source$(filesystem, fileId, version, creator, result)$
    /* `comment`**:** The file may be created for the first time by this process */
    if $(creator \neq p)$
        receive_file_transfer$(creator, fileId, version, data, piggyback)$
        send_acknowledgement$(creator, version.esn)$
        send_transfer_complete$(filesystem, fileId, version)$
        for all $e \in piggyback$
            $DetLog_p \leftarrow DetLog_p \cup \{\langle e, \{creator, p\}\rangle\}$
    $fce.fileId, fce.version, fce.data, fce.mode, fce.valid \leftarrow fileId, version, data, \texttt{WRITE}, \texttt{TRUE}$
    $FileCache_p \leftarrow FileCache_p \cup \{fce\}$
    $ESN_p \leftarrow ESN_p + 1$
    $fce.version.creator, fce.version.esn \leftarrow p, ESN_p$
    $DetLog_p \leftarrow DetLog_p \cup \{\langle version.creator, version.esn, p, ESN_p, \{p\}\rangle\}$
$fce.data \leftarrow data$

Figure 6.7: Process p executes $Write(fileId, data)$ during failure-free execution.

receive_acquire_ownership$(acquirer, fileId, version, mode)$

if $\exists fte \in FileTable$ **st** $(fte.fileId = fileId)$
    enqueue$(\langle acquirer, version, mode\rangle, fte.pendingQ)$
else if $(mode = \texttt{READ})$
    /* `comment`: The file does not exist */
    send_data_source$(acquirer, fileId, version, mode, \texttt{FALSE})$
else
    $fte.fileId, fte.cv, fte.versionSet, \leftarrow fileId, \perp, \emptyset$
    $fte.pendingQueue, fte.copySet \leftarrow \perp, \emptyset$
    enqueue$(\langle acquirer, version, mode\rangle, fte.pendingQueue)$
    $FileTable \leftarrow FileTable \cup \{fte\}$

Figure 6.8: File server enqueues ownership requests in pending queue of appropriate file.

while TRUE **do**
  select fte **from** $FileTable$ **st** $(\mathbf{head}(fte.pendingQ) \neq \perp)$
  $acquirer, version, mode \leftarrow \mathbf{head}(fte.pendingQ)$
  $\mathbf{dequeue}(fte.pendingQ)$
  if $(mode =$ READ$)$
    $GrantReadOwnership(acquirer, version, fte)$
  else if $(mode =$ WRITE$)$
    $GrantWriteOwnership(acquirer, version, fte)$

Figure 6.9: File server processes requests for ownership for the latest version of a file.

$GrantReadOwnership(acquirer :$ Agent$, version :$ VersionType$,$ **var** $fte :$ FileTableEntry$)$

  if $fte.mode =$ WRITE
    send_revoke_ownership$(fte.cv.creator, fte.fileId, fte.cv,$ WRITE$)$
    receive_ownership_revoked$(fte.cv.creator, fte.fileId, creator\_esn)$
    $fte.versionSet \leftarrow fte.versionSet \cup \{\langle fte.cv.creator, fte.cv.esn\rangle\}$
    $fte.cv.esn \leftarrow creator\_esn$
    $fte.mode \leftarrow$ READ
  send_data_source$(acquirer, fte.fileId, fte.cv, fte.cv.creator,$ TRUE$)$
  send_transfer_data$(fte.cv.creator, acquirer, fte.fileId, fte.cv)$
  receive_transfer_complete$(acquirer, fte.fileId, fte.cv)$
  $fte.copySet \leftarrow fte.copySet \cup \{acquirer\}$

Figure 6.10: Procedure $GrantReadOwnership$

$GrantWriteOwnership(acquirer : \text{Agent}, version : \text{VersionType}, \textbf{var}\, fte : \text{FileTableEntry})$

```
for all q ∈ fte.copySet
    send_revoke_ownership(q, fte.fileId, fte.cv, READ)
for all q ∈ fte.copySet
    receive_ownership_revoked(q, fte.fileId, fte.cv)
fte.copySet ← ∅
if fte.cv ≠ ⊥
    send_revoke_ownership(fte.cv.creator, fte.fileId, fte.cv, R&W)
    receive_ownership_revoked(fte.cv.creator, fte.fileId, creator_esn)
    fte.cv.esn ← creator_esn
    send_data_source(acquirer, fte.fileId, fte.cv, fte.cv.creator, TRUE)
    send_transfer_data(fte.cv.creator, acquirer, fte.fileId, fte.cv)
    receive_transfer_complete(acquirer, fte.fileId, fte.cv)
    fte.versionSet ← fte.versionSet ∪ {⟨fte.cv.creator, fte.cv.esn⟩}
    fte.cv.creator, fte.cv.esn ← acquirer, ⊤
else
    /* comment: The file is created for the first time */
    fte.cv.creator, fte.cv.esn, fte.versionSet ← acquirer, ⊤, ∅
    send_data_source(acquirer, fte.fileId, fte.cv, fte.cv.creator, TRUE)
fte.copySet ← {acquirer}
```

Figure 6.11: Procedure $GrantWriteOwnership$

```
receive_revoke_ownership(fileId, version, mode)
if version.esn = ⊤
    Let fce ∈ FileCache_p st (fce.fileId = fileId) ∧ fce.version is the latest version of fileId
else
    Let fce ∈ FileCache_p st ((fce.fileId = fileId) ∧ (fce.version = version))
esn ← fce.version.esn
if fce.mode = WRITE
    FileLog_p ← FileLog_p ∪ {⟨fce.fileId, fce.version, fce.data⟩}
if ((mode = READ) ∨ (mode = R&W))
    fce.valid ← FALSE
else if mode = WRITE
    fce.mode ← READ
send_ownership_revoked(p, fileId, esn)
```

Figure 6.12: Process p processes request to revoke its ownership of $fileId$.

receive_transfer_data$(acquirer, fileId, version, mode)$
Let $fle \in FileLog_p$ **st** $(fle.fileId = fileId) \wedge (fle.version = version)$
$piggyback \leftarrow \emptyset$
for all $e \in DetLog_p$ **st** $(|e.logged\_at| < f) \wedge (e.esn \leq version.esn) \wedge (acquirer \notin e.logged\_at))$
  $piggyback \leftarrow piggyback \cup \{\langle e.source, e.ssn, e.dest, e.esn\rangle\}$
send_file_transfer$(acquirer, fileId, fle.version, fle.data, piggyback)$

Figure 6.13: Process p processes request to transfer file $fileId$ to $acquirer$.

$Read(fileId$: FileId, $data$: DataType, $result$: Boolean)

  if $\exists \# e \in DetSet_p$ **st** $(\# e.desn = ESN_p + 1)$
    if $\exists fce \in FileCache_p$ **st** $((fce.fileId = fileId) \wedge (fce.valid = \texttt{TRUE}))$
      if (fce.mode = `WRITE`)
        $FileLog_p \leftarrow FileLog_p \cup \{\langle fce.fileId, fce.version, fce.data\rangle\}$
      $fce.valid \leftarrow \texttt{FALSE}$
    $data \leftarrow d.data$ where $d \in FileSet_p$ **st** $((d.fileId = fileId) \wedge (d.version = \langle \# e.source, \# e.sesn\rangle))$
    $fce.fileId, fce.version, fce.data, fce.mode, fce.valid \leftarrow fileId, d.version, data, \texttt{READ}, \texttt{TRUE}$
    $FileCache_p \leftarrow FileCache_p \cup \{fce\}$
    $result \leftarrow \texttt{TRUE}$
    $ESN_p \leftarrow ESN_p + 1$
  else if $(ESN_p \neq maxEsn_p)$
    $data \leftarrow fce.data$ where $fce \in FileCache_p$ **st** $((fce.fileId = fileId) \wedge (fce.valid = \texttt{TRUE}))$
    $result \leftarrow \texttt{TRUE}$
    $ESN_p \leftarrow ESN_p + 1$

Figure 6.14: Process p executes $Read(fileId, data, result)$ during roll-forward phase of recovery.

$Write(fileId : \text{FileId}, data : \text{DataType})$

$$
\begin{array}{l}
\quad \textbf{if } \exists \#e \in DetSet_p \textbf{ st } (\#e.desn = ESN_p + 1) \\
\quad\quad \textbf{if } \exists fce \in FileCache_p \textbf{ st } ((fce.fileId = fileId) \wedge (fce.valid = \texttt{TRUE}) \wedge (fce.mode = \texttt{WRITE})) \\
\quad\quad\quad FileLog_p \leftarrow FileLog_p \cup \{\langle fce.fileId, fce.version, fce.data \rangle\} \\
\quad\quad\quad fce.valid \leftarrow \texttt{FALSE} \\
\quad\quad fce.fileId, fce.mode, fce.valid \leftarrow fileId, \texttt{WRITE}, \texttt{TRUE} \\
\quad\quad fce.data \leftarrow d.data \quad \text{where } d \in FileSet_p \textbf{ st } ((d.fileId = fileId) \wedge (d.version = \langle \#e.source, \#e.sesn \rangle)) \\
\quad\quad fce.data \leftarrow data \\
\quad\quad ESN_p \leftarrow ESN_p + 1 \\
\quad\quad fce.version.creator, fce.version.esn \leftarrow p, ESN_p \\
\quad\quad FileCache_p \leftarrow FileCache_p \cup \{fce\} \\
\quad \textbf{else if } (ESN_p \neq maxEsn_p) \\
\quad\quad ESN_p \leftarrow ESN_p + 1 \\
\quad\quad fce.version.esn \leftarrow ESN_p \\
\quad\quad fce.data \leftarrow data
\end{array}
$$

Figure 6.15: Process p executes $Write(fileId, data)$ during roll-forward phase of recovery.

Chapter 7

Conclusion

The objective of our work is to address the fault-tolerance needs of non-critical distributed applications using techniques that impose a low-overhead on performance. We use rollback-recovery protocols such as checkpointing and message-logging as the foundation for our work. Though these protocols are well-studied (a survey paper on rollback-recovery [23] contains over 300 references to the literature on these protocols), these techniques have been rarely used for building reliable distributed applications. We believe that this is because of two reasons: (1) the few implementations of rollback-recovery protocols that are available do not facilitate configuring protocols for individual applications and (2) the performance of rollback-recovery protocols in practice is not well understood.

To make rollback-recovery protocols usable in practice for building reliable applications, we first design a toolkit for rollback recovery and then use it to provide a performance study.

In building our toolkit, because of the diversity in rollback-recovery protocols, the challenge is that it is necessary to emphasize the similarities that do exist among these protocols. To address this challenge, we develop a single unifying framework based on handling non-deterministic events in a process' execution. This framework allows us to characterize rollback-recovery protocols as event-driven programs which differ in their response to the same set of “relevant” events. We identify five classes of events that are relevant to all protocols and develop a simple specification language for expressing rollback-recovery protocols in terms of these events. We then implement *Egida*, an object-oriented toolkit for composing rollback-recovery protocols. Egida’s architecture is built around a set of “building blocks” that provide the functionality for handling the relevant events. From a specification, Egida can synthesize an implementation of the specified protocol by gluing together the appropriate objects from an available library of building blocks. Consequently, Egida facilitates rapid implementation of rollback recovery protocols with minimal programming effort.

To make Egida available to a large class of applications, we have integrated Egida as a middleware with the MPICH [29] implementation of the MPI (Message Passing Interface) standard 1.1. This enables existing MPI applications to take advantage of Egida without any modifications. Conversely, this allows us to evaluate the performance of rollback-recovery protocols using a large set of demanding applications.

We use Egida as a testbed to study some performance issues of rollback-recovery protocols. We identify limitations of protocols and develop novel solutions:

- Message-logging protocols do not simultaneously guarantee low-overhead during failure-free execution, fast recovery, and fault-containment. We develop a new class of *hybrid* protocols that for the first time combine causal logging with asynchronous receiver-based logging and provide these three desirable properties.

- Communication-induced checkpointing protocols are much less effective in practice than previously believed.

- Message-logging protocols impose a substantial overhead on performance whenever processes interact with the external environment, such as, the file system. To reduce this overhead, we develop a solution that is based on a more general approach, namely, treating parts of the environment as integral components that can be trusted to support failure recovery. We instantiate this approach by proposing a novel file-sharing protocol.

By building Egida and evaluating the performance of rollback-recovery protocols, we have taken some steps in making these protocols usable in practice. However, further steps need to be taken. For instance, handling other sources of non-determinism in an execution and enhancing Egida to provide fault-tolerance for multi-threaded applications present opportunities for future work.

Bibliography

[1] The NFS Distributed File Service. NFS White Paper, http://www.sun.com/software/white-papers/wp-nfs/, March 1995.

[2] L. Alvisi. *Understanding the Message Logging Paradigm for Masking Process Crashes*. PhD thesis, Cornell University Department of Computer Science, January 1996.

[3] L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and Orphan-Free Message Logging Protocols. In *Proceedings of the 23rd Fault-Tolerant Computing Symposium*, pages 145–154, June 1993.

[4] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, and Causal. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 229–236. IEEE Computer Society, June 1995.

[5] L. Alvisi and K. Marzullo. Tradeoffs in Implementing Optimal Message Logging Protocols. In *Proceedings of the Fifteenth Symposium on Principles of Distributed Computing*, pages 58–67. ACM, June 1996.

[6] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Transactions on Software Engineering*, 24(2):149—159, February 1998.

[7] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R.Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, February 1996.

[8] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 10–22, October 1992.

[9] R. Baldoni, F. Quaglia, and B. Ciciani. A VP-Accordant Checkpointing Protocol Preventing Useless Checkpoints. In *Proceedings of International Symposium on Reliable Distributed Systems*, pages 61–67, West Lafayette, IN, October 1998.

[10] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(2):47–76, February 1987.

[11] K. P. Birman and T. Joseph. Exploting Virtual Syncrony in Distributed Systems. In *Proceedings of the Eleventh Symposium on Operating System Principles*, pages 123–138, November 1987.

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

[13] D. Briatico, A. Ciuffoletti, and L. Simoncini. A Distributed Domino-Effect Free Recovery Algorithm. In *Proceedings of the IEEE International Symposium on Reliability Distributed Software and Database*, pages 207–215, December 1984.

[14] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Primary–Backup Protocols: Lower Bounds and Optimal Implementations. In *Proceedings of the Third IFIP Conference on Dependable Computing for Critical Applications*, September 1992.

[15] R. Butler and E. Lusk. Monitors, Message, and Clusters: The p4 Parallel Programming System. *Parallel Computing*, 20:547—564, April 1994.

[16] J. Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MIST: PVM with transparent migration and checkpointing. In *3rd Annual PVM Users' Group Meeting*, Pittsburgh, PA, May 1995.

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

[18] 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.

[19] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, October 1996.

[20] O. P. Damani and V. K. Garg. How to Recover Efficiently and Asynchronously when Optimism Fails. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 108–115, 1996.

[21] D. Dolev and D. Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4), April 1996.

[22] E. N. Elnozahy. On the relevance of communication costs of rollback-recovery protocols. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 74–79, August 1995.

[23] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999.

[24] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The Performance of Consistent

Checkpointing. In *Proceedings of the Eleventh Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

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

[26] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *Digest of Papers: 24 Annual International Symposium on Fault-Tolerant Computing*, pages 298–307. IEEE Computer Society, June 1994.

[27] M. Norman et. al. Galaxies collide on the I-WAY: An example of heterogeneous wide-area collaborative supercomputing. *International Journal of Supercomputer Applications*, 10(2):131—140, 1996.

[28] J. N. Gray. Notes on Data Base Operating Systems. In R. Bayer, R. M. Graham, and G. Seegmueller, editors, *Operating Systems: An Advanced Course*, pages 393–481. Springer-Verlag, 1977. Lecture Notes on Computer Science 60.

[29] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[30] J. M. Hélary, A. Mostefaoui, R. H. B. Netzer, and M. Raynal. Preventing Useless Checkpoints in Distributed Computations. In *Proceedings of IEEE International Symposium on Reliable Distributed Systems*, pages 183–190, 1997.

[31] J. M. Hélary, A. Mostefaoui, and M. Raynal. Virtual precedence in asynchronous systems: concepts and applications. In *Proceedings of the 11th Workshop on Distributed Algorithms*. LNCS press, 1997.

[32] M. A. Hiltunen and R. D. Schlichting. A Configurable Membership Service. *IEEE Transactions on Computers*, 47(5):573—586, May 1998.

[33] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[34] Y. Huang and C. Kintala. Software Implemented Fault Tolerance: Technologies and Experience. In *Proceedings of the IEEE Fault-Tolerant Computing Symposium*, pages 2–9, 1993.

[35] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[36] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons Inc., 1991.

[37] D. B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. PhD thesis, Rice University, December 1989. Available as report COMP TR89-101.

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

[39] D. B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. In *Digest of Papers: 17 Annual International Symposium on Fault-Tolerant Computing*, pages 14–19. IEEE Computer Society, June 1987.

[40] D. B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11:462–491, 1990.

[41] T. Y. Juang and S. Venkatesan. Crash recovery with little overhead. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 454–461. IEEE Computer Society, June 1987.

[42] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[43] T. H. Lai and T. H. Yang. On Distributed Snapshots. *Information Processing Letters*, (25):153–158, 1987.

[44] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[45] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133—169, May 1998.

[46] C. Lee, C. Kesselman, and S. Schwab. Near-realtime satellite image processing: Metacomputing in CC++. *IEEE Computer Graphics and Applications*, 16(4):79—84, 1996.

[47] A. Lenstra. Factoring integers using the Web and the number field sieve. Technical report, Bellcore, August 1995.

[48] J. Leon, A. L. Ficher, and P. Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, February 1993.

[49] C. C. Li and W. K. Fuchs. CATCH: Compiler-assisted techniques for checkpointing. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, pages 74–81, 1990.

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

[51] D. Lowell and P. M. Chen. Discount Checking: Transparent, Low-Overhead Recovery for General Applications. November 1998.

[52] D. Manivannan and M. Singhal. A Low-Overhead Recovery Technique Using Quasi Synchronous Checkpointing. In *Proceedings of International Conference on Distributed Computing Systems*, pages 100–107, 1996.

[53] F. Mattern. Virtual Time and Global States of Distributed Systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms*, pages 215–226. Elsevir Science Publishers B. V., 1989.

[54] J. R. Mitchell and V. K. Garg. A Non-Blocking Recovery Algorithm for Causal Message Logging. In *Proceedings of the 17th Symposium on Reliable Distributed Systems*, pages 3–9, West Lafayette, IN, October 1998.

[55] G. Muller, M. Hue, and N. Peyrouz. Performance of consistent checkpointing in a modular operating system: Results of the FTM experiment. In *Proceedings of the First European Dependable Computing Conference (EDCC-1)*, pages 491—508, October 1994.

[56] R. Netzer and J. Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. Technical Report 93-32, Department of Computer Sciences, Brown University, July 1993.

[57] N. Neves and W. K. Fuchs. RENEW: A Tool for Fast and Efficient Implementation of Checkpoint Protocols. In *Proceedings of the 28th IEEE Fault-Tolerant Computing Symposium (FTCS)*, Munich, Germany, June 1998.

[58] J. Nieplocha and R. Harrison. Shared memory NUMA programming on the I-WAY. In *Proceedings of Fifth IEEE Symposium on High Performance Distributed Computing*, pages 432—441, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[59] J. S. Plank, M. Beck, and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems Newsletter, Special Issue on Fault Tolerance*, pages 62—67, December 1995.

[60] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt:Transparent checkpointing under Unix. In *Proceedings of the USENIX Technical Conference*, pages 213–224, January 1995.

[61] J. S. Plank and K. Li. Faster checkpointing with $N + 1$ parity. In *Proceedings of the Twenty Fourth International Symposium on Fault-tolerant Computing (FTCS-24)*, pages 288—297, June 1994.

[62] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science*, pages 46–57, November 1977.

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

[64] B. Ramamurthy, S. J. Upadhyaya, and R. K. Iyer. An Object-Oriented Testbed for the Evaluation of Checkpointing and Recovery Systems. In *Proceedings of the 27th IEEE Fault-Tolerant Computing Symposium*, pages 194–203, June 1997.

[65] B. Randell. System Structure for Software Fault-Tolerance. *IEEE Transaction on Software Engineering*, SE-1(2):220–232, June 1975.

[66] R. Van Renesse, K. Birman, B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A Flexible Group Communications System. Technical Report TR95-1500, Cornell University Computer Science Department, March 1995.

[67] R. Van Renesse, T. Hickey, and K. Birman. Design and Performance of Horus: A Lightweight Group Communications System. Technical Report TR94-1442, Cornell University Computer Science Department, August 1994.

[68] M. Rosenblum and J. K. Ousterhout. A Case for Log-Structured File Systems. In

Proceedings of the Thirteenth Symposium on Operating Systems Principles, pages 1–15, Pacific Grove, CA, October 1991. ACM.

[69] D. Schmidt, D. Box, and T. Suda. Adaptive: A Dynamically Assembled Protocol Transformation, Integration, and Evaluation Environment. *Concurrency: Practice and Experience*, 5(4):269—286, June 1993.

[70] F. B. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems*, 4(2):125—148, 1982.

[71] F. B. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[72] F. B. Schneider. Paradigms for distributed programs. In *Distributed Systems—Methods and Tools for Specification*, volume 190. Springer-Verlag, New York, NY, 1985.

[73] F. B. Schneider. Implementing fault–tolerant services using the state machine approach: A Tutorial. *Computing Surveys*, 22(3):299–319, September 1990.

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

[75] A. Singhai, A. Sane, and R. Campbell. Quarterware for Middleware. In *Proceedings of the International Conference on Distributed Computing and Systems (ICDCS'98)*, pages 192—201, Amsterdam, Holland, May 1998.

[76] A. P. Sistla and J. L. Welch. Efficient Distributed Recovery Using Message Logging. In *Proceedings of the Eighth Symposium on Principles of Distributed Computing*, pages 223–238. ACM SIGACT/SIGOPS, August 1989.

[77] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of the International Parallel Processing Symposium*, pages 526–531, April 1996.

[78] R. B. Strom and S. Yemeni. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, April 1985.

[79] R. E. Strom, D. F. Bacon, and S. A. Yemini. Volatile Logging in n-Fault-Tolerant Distributed Systems. In *Proceedings of the Eighteenth Annual International Symposium on Fault-Tolerant Computing*, pages 44–49, 1988.

[80] S. Venkatesan and T. Y. Juang. Efficient Algorithms for Optimistic Crash Recovery. *Distributed Computing*, 8(2):105–114, June 1994.

[81] Y. M. Wang, O. P. Damani, and V. K. Garg. Distributed Recovery with K-Optimistic Logging. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 60–67, 1997.

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

[83] Y. M. Wang, Y. Huang, K. P. Vo, P. Y. Chung, and C. Kintala. Checkpointing and Its Applications. In *Proceedings of the IEEE Fault-Tolerant Computing Symposium (FTCS-25)*, pages 22–31, Pasadena, CA, June 1995.

[84] Y. M. Wang and W. J. Lee. COMERA: COM Extensible Remoting Architecture. In *Proceedings of the 4th Conference on Object Oriented Technologies and Systems (COOTS)*, April 1998.

[85] J. Xu and R. H. B. Netzer. Adaptive Independent Checkpointing for Reducing Rollback Propagation. Technical Report CS-93-25, Brown University, May 1993.