Lightweight Fault-tolerance for Highly Cooperative Distributed Applications

Lorenzo Alvisi, **Sriram Rao**, and **Harrick M. Vin**

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

Abstract

The recent introduction of high-speed networks, faster processors, and the rapid growth of heterogeneous large-scale distributed systems has enabled the development of distributed applications that move beyond the client-server model to truly harness the computational potential of distributed systems. These new applications will be structured around groups of agents that communicate using messages as well as files. Some of these emerging applications will be critical enough to life or business to warrant explicit process replication to achieve high availability. Often, however, explicit replication will be too costly to implement, or, simply, high availability will not be necessary. In these circumstances, the availability of low-overhead fault-tolerance techniques will be crucial to achieving reliability. To address these needs, we are developing lightweight fault-tolerance (LFT), a new low-overhead approach to fault-tolerance for highly cooperative distributed applications.

In the first part of this paper, we describe how LFT extends to file communication the causal logging techniques used in message passing. We show how in our approach all the synchronous operations that are currently performed by log-based protocols during file I/O are either eliminated or made asynchronous, therefore removing the opportunities for blocking. Furthermore, we argue that our approach has the potential to enhance the effectiveness of existing rollback recovery techniques for software fault-tolerance. In the second part of the paper, we validate LFT through extensive simulation. Our results indicate that LFT brings the cost of file communication down to the level of message passing, drastically reducing the overhead incurred by fault-tolerant applications in performing file I/O.

1 Introduction

High-speed networks, faster processors, and the rapid growth of heterogeneous large-scale distributed systems are enabling a new class of highly cooperative distributed applications that move beyond the client-server model to harness the computing potential of distributed systems. These applications will be structured around groups of agents that, depending on the nature of their interaction, will communicate in different ways. For instance, tightly-coupled agents will use message passing—either directly or through distributed shared memory—to achieve low-latency and high bandwidth. However, message passing can be inefficient when agents are loosely-coupled, or when the precise identity of the intended receiver is unknown to the sender. For loosely-coupled agents communication through files will be more appropriate.

Some of these emerging applications will be critical enough to life or business to warrant explicit process replication to achieve high availability. Often, however, explicit replication will be too costly to implement, or high availability will simply not be necessary. In these circumstances, the availability of low-overhead fault-tolerance techniques will be crucial to achieving reliability.

To address these needs, we are developing *lightweight fault-tolerance* (LFT), a new low-overhead approach to fault-tolerance for highly cooperative distributed applications. Lightweight fault-tolerance has the following goals:

- To require few additional resources and have a negligible impact on performance during failure-free executions.
- To integrate with applications in a way that is transparent to the application programmer.
- To scale the cost of providing fault-tolerance depending on the severity and number of failures that need to be tolerated.
- To support and enable applications in which communication occurs through both messages and files.

Many of these goals are shared by other low-overhead fault-tolerance techniques and have already been addressed with considerable success. In designing our solution, we build on some of these techniques. In particular, LFT uses rollback-recovery to minimize dedicated resources, and causal logging [1, 2, 3, 10] to minimize the impact on application performance and to scale cost with the number of failures that need to be tolerated. Furthermore, transparency is achieved by implementing LFT as a middleware.

LFT is unique in its focus on efficient support of fault-tolerance for applications in which communication occurs through both messages and files. Other techniques based on rollback recovery—such as primary-backup, checkpointing, and message logging—may become expensive when performing file I/O. Since the file system is considered to be a component of the external environment, the application may have to block for every file I/O operation while data critical to recovery are logged to stable storage. For their part, toolkits such as Isis [6], Horus [20] and Transis [9] assume that all communications occurs through message passing and treat the file systems as a potentially dangerous *hidden channel* of communication through which causal dependencies are not tracked. In contrast, LFT presents the file system to the application as an integrated

partner that can be trusted to provide the data needed during recovery and not as a detached component of the external environment.

In the first part of this paper, we describe how LFT extends the causal logging techniques used in message passin to file communication. We show how in our approach all the synchronous operations that are currently performed by log-based protocols during file I/O are either eliminated or made asynchronous, therefore removing the opportunities for blocking. Furthermore, we argue that our approach has the potential to enhance the effectiveness of existing rollback recovery techniques for software fault-tolerance.

In the second part of the paper, we validate LFT through extensive simulation. Our results indicate that LFT brings the cost of file communication down to the level of message passing, drastically reducing the overhead incurred by fault-tolerant applications in performing file I/O.

The remainder of the paper is organized as follows. Section 2 describes our system model. The ideas and methodologies at the basis of lightweight fault-tolerance are presented in Section 3. Section 4 describes our simulation, and contains an analysis of its results. Section 5 concludes the paper.

2 System Model

We assume an asynchronous distributed system, in which there exist no global time source, no bound on the relative execution speed of agents, and no bound on transmission delays. We model agents as processes, and in the rest of the paper we use the two terms interchangeably. Processes communicate using both messages and files. The execution of a process is represented as a sequence of send, receive, read, write, and local events. 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 from process p_1 to process p_2, we assume that p_2 delivers m only if it has received m and that p_2 delivers m at most once. Furthermore, we assume that a correct process will eventually deliver all messages it has received.

At any point in time, 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.

Execution of a process is *piecewise deterministic* [26]: It consists of a sequence of deterministic intervals of execution, joined by non-deterministic events. For each process, the first interval of execution begins with the process' initial state; subsequent intervals begin with each non-deterministic event. Hence, execution of a process consists of a sequence of intervals, the beginning of each interval being defined by the initial state of the process and by the non-deterministic event. Such intervals are called a *state intervals*. Given the first state of a state interval and the deterministic event that defines the beginning of the interval, the remaining states in the interval are uniquely determined. We assume only two kinds of non-deterministic events:

`deliver events:` when an agent delivers a message, it chooses the message nondeterministically among those received by the communication sub-system but not yet presented to the application.

`read events`: when an agent reads a file, the version of the file read by the process is non-deterministic.

Processes exchange messages over point-to-point and FIFO channels that can fail by transiently losing messages. Finally, processes themselves can fail according to the fail-stop model [22]. Hence, we assume that processes fail independently, only by halting, and that a faulty process is eventually detected by all correct processes.

2.1 Consistency

Since we are interested in addressing applications that communicate through both messages and files, our notion of consistency must capture both styles of communication.

For message-based communication, given the states s_p and s_q of two processes p and q, $p \neq q$ respectively, we say that s_p and s_q (or, more simply, p and q) are *mutually message-consistent* if all of the 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.

To define a similar notion for file-based communication, we first observe that, since the content of a file may change as a result of a write event, at any point after its creation a file has a unique *version* v. Given a file f, we denote its version by $f.v$. Then, given the states s_p and s_q of two processes p and q, $p \neq q$ respectively, we say that s_p and s_q (or, more simply, p and q) are *mutually file-consistent* when, for all versions v and files f, if p has read during its execution up to s_p file $f.v$ written by q, then q has written $f.v$ during its execution up to s_q.

If two processes p and q are both mutually message-consistent and mutually file-consistent, then we say that p and q are *mutually consistent*. Finally, a collection of states, one from each process, is a *consistent global state* if all pairs of states are mutually consistent [8]; otherwise it is *inconsistent*.

3 Lightweight Fault-Tolerance

To minimize explicit process replication, fault-tolerance in LFT is achieved through log-based rollback recovery. Each process p periodically records its local state on stable storage in a checkpoint. Furthermore, p saves enough information about each non-deterministic event e executed since the last checkpoint to guarantee that, after executing e during recovery, p will again enter the same state entered during the original execution. We call such information the *determinant* of e, and we say that a determinant is *stable* if it can not be lost as a result of a failure. If p crashes, recovery involves (1) creating a new instance of process p, (2) initializing p to the latest checkpointed state, and (3) restarting p, making sure to repeat each non-deterministic event according to the information saved in the corresponding determinant. The policy used in determining when and where determinants are to be logged coordinates with the recovery protocol to guarantee that upon recovery the global state of the system is consistent. Such consistency is often expressed in terms of *orphan processes*—surviving processes whose state is inconsistent with the recovered state of a crashed process. Consistency for log-based protocols translates into the guarantee that upon recovery no process is an orphan.

The performance of a logging-based protocol depends heavily on how the protocol enforces the no-orphans guarantee. For instance, pessimistic protocols (for example, [7, 19, 13, 26]) never create orphans by logging determinants on stable storage synchronously. Unfortunately, these protocols exhibit relatively poor performance during failure-free runs, since they prevent processes from communicating until logging is complete. In contrast, optimistic protocols protocols [25, 23, 14, 27]) log determinants asynchronously and do not delay communication, thus achieving good failure-free performance. Unfortunately, these protocols may create orphans, and hence may force correct processes to roll back in order to reconstitute a consistent global state.

LFT is based on a third logging technique called *causal logging* [2]. Causal logging protocols implement the no-orphans guarantee by enforcing the following *CL property*: if the state of a process p causally depends [15] on a non-deterministic event e, then either the determinant of e is stable, or p has a copy of e's determinant in its volatile memory. In either case, all determinants needed during recovery to restore the system in a global state consistent with p's state are available. Hence, p will never become an orphan.

Causal logging protocols [1, 10, 2, 3] combine the positive aspects of pessimistic and optimistic protocols. Causal protocols do not log determinants synchronously, and thus achieve the performance advantages of optimistic protocols. At the same time, causal protocols never create orphans, and thus achieve the fault-containment advantages of pessimistic protocols.

In the next section, we briefly describe family-based logging, the causal logging implementation used by LFT for applications that communicate through message passing. We then present the limitations of log-based techniques in general, and family based logging in particular, when applied directly to applications in which processes communicate both through files and messages. Finally, we discuss how family-based logging can be extended to achieve lightweight fault-tolerance for such applications.

3.1 Family-Based Logging

Family-based logging (FBL) [1, 3] is a low-overhead implementation of causal logging. As a causal protocol, FBL never delays communication (except when communicating with the external environment), and at the same time never creates orphans. Furthermore, FBL allows determinants to be maintained in the processes' volatile memory, and does not require processes to send any additional message over those needed to mask transient link failures. Finally, FBL protocols can be tuned so that their overhead (1) depends on the number of failures that an application is willing to tolerate, and (2) can be minimized by exploiting the pattern of inter-process communication exhibited by a specific application [3].

Family-based logging protocols exploit the observation that, given a maximum number f of concurrent failures that are to be tolerated, a determinant is stable once it is replicated in the volatile memory of $f + 1$ processes that fail independently. Hence, in FBL the CL property is implemented by having each process p piggyback on its messages all non-stable determinants p previously generated, plus all non-stable determinants that had been piggybacked on messages previously received by p.

Central to FBL's performance is limiting the size of the piggybacked information. Fortunately, empirical evidence shows that the number of determinants piggybacked does not grow uncontrollably [10, 1].

Furthermore, since we assume that the only non-deterministic events in the system are delivery events, determinants are small. The determinant associated with the delivery of a message m is the tuple: $\langle m.source, m.ssn, m.dest, m.rsn \rangle$, where $m.source$ and $m.dest$ denote, respectively, the identity of the sender process and of the destination process; $m.ssn$—message m's *send sequence number*—is a unique identifier assigned to m by the sender $m.source$; and $m.rsn$—message m's *receive sequence number*—represents the order in which m was delivered: $m.rsn = \ell$ if m is the ℓ^{th} message delivered by m's destination process.[25].

Note that the text $m.data$ of m is not part of the determinant. Replicating $m.data$ is not necessary because if the determinants of all the messages delivered during an execution are available then it is always possible during recovery to regenerate any message m that was sent and delivered during the original execution. As we discuss next, the structure and management of determinants becomes more complex when we move from messages to files.

3.2 The gap between applications and file-system

Techniques that introduce little overhead when applied to message passing applications become considerably more costly when communication also involves file I/O. In particular, logging-based techniques may delay processes for every file I/O operation while data critical to recovery are logged on stable storage. This is true even for techniques such as family-based logging, that successfully avoid delays when applied to pure message-based communication.

We believe that the main cause for this lack of performance is the gap that currently exists between applications and the file system when it comes to fault-tolerance. This gap causes fault-tolerance techniques designed for message passing applications to exhibit a somewhat schizophrenic pattern in their interactions with the file system. On the one hand, these techniques treat the file system as a partner on which they rely to provide stable storage for checkpoints and other information used during recovery. On the other hand, when reading or writing other kinds of files, the file system is treated as a generic component of the external environment that cannot be trusted to support recovery after a failure. Specifically, applications do not assume that input provided by the environment will be reproducible by the environment during recovery, and furthermore do not assume that, if a failure occurs in the system, the external environment can roll back to a previous state in order to become consistent with the recoverable state of the application.

This attitude towards the file system has multiple serious negative consequences for the log-based fault-tolerance techniques that are the focus of our attention.

A first negative consequence is loss of performance, as interaction with the external environment is a major source of overhead for log-based protocols [11]. In particular, treating the file system as a component of the external environment requires applications to take the following steps whenever reading or writing a file.

On reads: File read events are non-deterministic events (see Section 2). Hence, to guarantee recovery file read events must be logged in stable storage. The determinant of a read event has the following

form: $\langle name, v, reader, data \rangle$, where *name* is the file's name, v is the file's version, *reader* is the identity of the reader process and *data* is the file's data.

As opposed to message determinants, file determinants include explicitly the file's content, which is not assumed to be reproducible. Hence, since determinants can be large, it becomes impractical to use piggybacking schemes, such as those employed in FBL, that implement stable storage through replication in volatile memory. In practice, determinants are instead logged using a version of stable storage implemented by the file system. Furthermore, read determinants must be logged synchronously. In particular, the reader must delay sending messages or writing files until logging of any read determinant has completed. This delay is necessary, since the file system does not guarantee that any given file version will be available for replay if the reader fails.

On writes: Since the file system in general cannot roll back, output must be delayed until the state in which the write is generated is guaranteed to be recoverable. This guarantee is achieved by executing an *output commit* protocol, which synchronously writes on stable storage the information needed to recover. In causal logging protocols, the output commit consists of synchronously flushing to stable storage the non-stable determinants kept in the volatile memory of the process communicating with the external environment—in this case, the writer process.

A second less obvious negative consequence is that tolerating software generated failures becomes more problematic. Most of the software bugs that survive through design reviews, quality assurance, and beta testing manifest themselves under transient system conditions that are very difficult to reproduce—the elusiveness of these bugs has gained them the name of *Heisenbugs* [12]. Experience shows that a very effective way to handle Heisenbugs is to roll back the faulty process to an earlier state and then to restart execution. The earlier the state a process is rolled back to, the more likely the process is to follow an execution that is sufficiently different from the original one to avoid the Heisenbug. Unfortunately, since the file system is considered part of the external environment, a process can never roll back past the last state in which it performed a write operation. Hence, frequent writes to the file system can limit the effectiveness of these rollback-based techniques, since they limit the extent by which a process can roll back.

Finally, another negative consequence of the gap between applications and the file system is that, as much as applications do not rely on the file system as a fully trusted partner in their recovery, distributed file systems do not rely on the clients' ability to tolerate failures to improve their own reliability and performance. For instance, several distributed file systems [5, 28, 16, 17, 18, 24] rely on recovery protocols that use information provided by clients to restore the state of a faulty server's cache. These protocols for server recovery, however, are not designed for environments in which (1) clients cooperate in distributed applications and (2) clients attempt to recover to a consistent state after a system failure. It is easy to show that in such an environment, if just one client fails concurrently with the server, then the protocols for server recovery may recover the server in an inconsistent state, potentially requiring all clients to reboot.

As another example, consider the policy that is currently enforced by most distributed file systems in order to regulate file sharing. Suppose process p creates a new version v of file f. If p delays writing $f.v$

to the file server and another process q requests to read f, then the server requires p to synchronously write back $f.v$ before q is allowed to read it. Although these synchronous writes are expensive, the file system enforces them to keep q from reading inconsistent data. In fact, if p were to transfer $f.v$ directly to q without first writing the file back and then p crashes, then the file system would not be able to guarantee that $f.v$ would be regenerated.

3.3 Lightweight fault-tolerance for file-based communication

The goal of LFT for file-based communication is to bridge the gap between application-level fault-tolerance software and the file system and, in so doing, to address the negative effects that we have described in the previous section. In particular, LFT is designed to achieve the following results.

1. To bring the cost of communicating through files to the same low level as the cost of communicating through messages.

2. To enhance the effectiveness of existing rollback recovery techniques for software fault-tolerance.

3. To free file systems from the fault-tolerance-driven considerations that have so far prevented the implementation of a high performance, truly server-less file system.

Clearly, to achieve these results LFT must eliminate the synchronous actions that are currently executed for each file I/O operation, i.e. (1) the synchronous logging of a file content that occurs when a file is read and (2) the synchronous execution of an output commit protocol that occurs when a file is written. We first focus on eliminating the synchronous logging that occurs on reads.

The reason for logging the content of the files read by the application is to guarantee that these files will be available for replay in case of a system failure. In particular, a process that issues a read operation cannot distinguish during recovery if the file being read comes from the file system or from the log kept by the fault-tolerance software. In effect, through file logging and replay, current fault-tolerance techniques replicate some of the functionalities of the file system.

A central thrust of LFT is to avoid such duplication of functionality. To achieve this result, LFT extend the file system to support *file versioning*. In LFT, the file system keeps (at least conceptually) all versions of the files created during an application's execution. File versioning is transparent to the application, which continues to interact with the file system in the usual way. After a system failure, our fault-tolerance middleware cooperates with the file system to guarantee that a recovering process read the same file versions the process read before crashing.

An obvious advantage of file versioning is that it eliminates the need for synchronously logging file contents to stable storage for each read operation. A second, perhaps less obvious, advantage of file versioning is that it removes $f.data$ from the determinant of a read event for file f. The importance of this second advantage becomes clear when we consider that maintaining the file versions is useless unless the determinants of the read events are available during recovery. Determinants must be stored on stable storage, since they contains the information necessary to decide during recovery which process read a specific version

of a given file. Fortunately, since now the file's data is not part of a read determinant, the sizes of read determinants and message delivery determinants are comparable. Hence, we can provide stable storage for read determinants by using the same piggybacking techniques that we used in family-based logging to replicate message determinants in volatile memory. Since these techniques do not introduce blocking, versioning effectively eliminates all potentially blocking actions that are currently performed on a file read.

We now consider how to eliminate the execution of a synchronous output commit protocol whenever writing a file. We observe that in most cases there is no reason for the output commit associated with a write operation to be synchronous. Writes to the file system are often delayed and batched to achieve better performance. Hence, unless file sharing or other considerations force the write to the file-system to be synchronous, the output commit protocol can be performed in the background. The only requirement then becomes that output commit should complete before the corresponding write.[1]

This observation eliminates in many cases the performance cost of executing synchronous output commit protocols for writes. However, it does not address the issue of enhancing the effectiveness of rollback techniques for transient software failures. In fact, output commits are still performed, although often asynchronously. Hence, if writes are frequent, applying rollback recovery techniques to mask Heisenbugs is still problematic.

To address this concern, we need to completely eliminate output commits associated with write events, allowing the file system to roll back past its last write to the file system. Fortunately, in LFT it is fairly easy to allow the file system to roll back. For instance, LFT could associate with each version v of a file f the notion of a *creator* process, whose identity can be for instance encoded in the version field v of the determinants corresponding to read events for file $f.v$. When a process p requests to read a file f, the file system would inform the LFT layer, which would contact the creator c of the last version $f.v$ of f. If it had not done so before, process c would synchronously write $f.v$ to the file server. Process c would then send to p those determinants that c had in its volatile memory when it created $f.v$ Concurrently, either the file server or c itself would send $f.v$ to p. Process p would be allowed to read $f.v$ only after having received the determinants from c.

The scheme that we have just outlined eliminates output commits for write operations and makes it easier to mask transient software failures. However, it is not fully satisfactory. First, while the scheme eliminates output commit at the writer side, it requires the reader to wait for the delivery of the determinants from c before accessing $f.v$. Second, when $f.v$ is shared, process c is forced to synchronously write $f.v$ to the file server.

We focus our attention to the second of these problems, since by solving the second problem the first would disappear too: if c were allowed to send directly $f.v$ to p without first writing it back to the file server, then c could just piggyback the determinants onto $f.v$, and p would not have to wait.

Note that the policy of writing back dirty data to the file server before it can be shared is not an artifact of the use of LFT, but rather it is commonly adopted in current file systems. In these systems, data is stable

[1] Although this optimization is not specific to LFT, and for instance can be applied to any log-based fault-tolerance techniques we are unaware of any technique that takes advantage of it.

only when it is written to the file server. Hence, process c is not allowed to transfer $f.v$ directly to process p because, if c were to crash before having made $f.v$ stable by writing it to the file server, then $f.v$ would be lost. Fault-tolerance then becomes a bottleneck in the development of the truly server-less high performance file systems of the future [4].

We believe that LFT can result in a significant step towards the elimination of this bottleneck. As we argue below, with LFT $f.v$ does not need to be written to the file server in order to be stable. Hence, c can directly pass the data to p without having to incur the cost of a synchronous write to the server and can later write f to the file server asynchronously. Thus, LFT effectively makes communicating through files as cheap as communicating through messages.

This result is possible because, by using family-based logging for the determinants of both read and deliver events, LFT guarantees that no orphan processes are created as long as the number of concurrent failures is within the limit allowed by the FBL protocol. Hence, under the assumption that processes are piecewise deterministic, LFT can regenerate during recovery the content of any version of a file that was read by a correct process, in a way similar to what we discussed for message contents in Section 3.1.

LFT trades-off performance during failure-free performance for fast recovery. If a version $f.v$ is not synchronously written to the file server, if c and p concurrently fail after p has read f, then during recovery p has to wait for c to regenerate f. However, if failures are rare, we feel that allowing c to write $f.v$ asynchronously is highly preferable. Furthermore, if fast recovery of p is a concern, LFT can be instructed to log $f.v$ in p's local disk. Note however that this logging is once again performed asynchronously.

4 Experimental Evaluation

To quantify the cost of achieving fault-tolerance using existing approaches as well as to demonstrate the efficacy of LFT, we have carried out extensive simulations. In what follows, we describe our simulation environment and the results of our simulations.

4.1 Simulation Environment

We consider a distributed application consisting of N processes that interact with each other by exchanging messages as well as through a file server (i.e., by sharing files). Each instruction executed by a process is either a *computational* (i.e., arithmetic or control flow instruction), a *message passing* (i.e., Send and Receive), or a *file I/O* (i.e., Read and Write) instruction. For simplicity, we assume that all the computational instructions execute in the same amount of time and do not impose any fault-tolerance overhead. The execution time for message passing and the file I/O instructions, on the other hand, consists of two components: (1) the time to execute the operation in a fault-free environment (e.g., the time to read a file from an NFS server, the blocking delay incurred while waiting for a message, etc.), and (2) the overhead (yielded by logging, output commit, etc.) for achieving fault-tolerance. Thus, a program's execution time is dependent both on the instruction mix as well as the time required to execute each instruction.

To compute the effect of various fault-tolerance techniques on the execution times, we have developed an event-driven simulator. In our simulator, the instructions executed by each process are governed by

three parameters: P_c, P_m, and P_f, which denote the fractions of computational, message passing, and file I/O instructions constituting a program, respectively. Observe that $P_c + P_m + P_f = 1$, and that the ratio $\frac{P_m}{P_f}$ determines the extent to which the program is message passing or file I/O oriented. For each process, the next instruction to be executed is determined by the simulator in accordance with the values of P_c, P_m, and P_f using an independently seeded random number generator. If the next instruction is a computational instruction, then the total execution time for the process is appropriately incremented, and the execution continues. If the next instruction is a message passing instruction, then the simulator first determines whether the instruction is a Send or a Receive (for simplicity, we assume that Send and Receive instructions are equally likely). For a Send instruction, the simulator selects a process at random and then sends a message. On the other hand, when a process (say r) executes a Receive instruction, the simulator selects a process (say s) at random, and determines if waiting for a message from s will result in a deadlock. If a deadlock is detected, then the simulator selects another process at random, and the procedure repeats. If waiting for a message from s does not result in a deadlock, then the simulator sends a request for a message to s (containing $r.inst$, the number of instructions executed by process r prior to executing the Receive operation) and blocks process r. To ensure that process r receives a response from s within a finite amount of time, we assume, without loss of any generality, that process s responds to process r's request as soon as the number of instructions executed by process s exceeds that of r (i.e., when $s.inst \geq r.inst$).

Finally, to support file I/O operations, we simulate a file server that enforces a *single writer* policy, whereby, at any given instant, at most one client is allowed to write to a file. Multiple clients are allowed to read a file concurrently and each read sees the effects of all the previous writes. The server supports client caching of data and a delayed write-back policy (i.e., the data is written to the server only when a process explicitly flushes its cache or when some other process requests the information), and uses callbacks to inform clients when their cached data is to be invalidated.

For purposes of message logging, we assume that the processes employ the family-based message logging protocol. Moreover, to ensure that failure recovery information is available even in the presence of processor failure, we assume that the fault-tolerance information that must be stored on stable storage is logged at the file server[2].

4.2 Performance Measurements and Metric

To quantify the reduction in the fault-tolerance overhead yielded by various techniques presented in Section 3, we have simulated the following three fault-tolerance schemes:

- EXISTING: This scheme reflects the state-of-the-art of logging-based techniques, in which applications treat the file system as a generic component of the external environment. As per this scheme:

[2]Observe that a log can also be maintained on the local disks at a client site. Although potentially more efficient, such a the log may be unavailable for an extended period of time during processor failures and therefore may preclude re-starting the process on another machine. Hence, for the remainder of this section, we will only present the results for the scenario in which the file server is used as the stable storage.

1. A Send operation results in logging to stable storage: (1) the non-stable determinants resulting from prior message passing operations, and (2) all the file information read since the previous Send or Write.
2. A Write operation results in the invocation of the synchronous output commit to the file server. This involves writing to the file server: (1) any of the non-stable determinants (yielded by message passing operations) being maintained in volatile memory of the writer, and (2) all the file information read since the previous Send or Write. The data, on the other hand, is written to the server using the delayed write-back policy.
3. The Read and Receive operations block the process until the requested information is received.

- VERSIONING: In this scheme, the Read and Receive operations are exactly the same as in the EXISTING approach. The fault-tolerance overhead yielded by the Send and Write operations, however, is reduced using the following techniques:

  1. The file server maintains multiple *versions* of each file, and hence, can be trusted to provide clients with the correct version of the data during recovery. Consequently, the Send operation is not required to log any of the file data read since the previous Send or Write operations.
  2. For the Write operation, the output commit is performed *asynchronously*. That is, the process executing the Write operation initiates a thread to perform the output commit operation, and then continues execution. The data is written to the disk using the delayed write-back policy. Thus, when a process p executes a Read or a Write operation of file f, and if process q currently holds the most recent version of file f, then process p must wait until: (1) the asynchronous output commit initiated by q as a result of its most recent Write to file f completes, and (2) the content of file f at the server becomes consistent with process q.

- LFT: This scheme implements all of the optimizations described in Section 3. Specifically, it improves upon the VERSIONING scheme by completely eliminating the output commit and by enabling processes to pass around dirty data along with the file determinants. This optimization is implemented as follows: When a process p requests the content of a file f, the file server forwards the request to the process (say q) that is currently holding the most recent version of file f. Process q then *directly* passes the data from its cache along with the determinants to process p.

To separate the overhead of fault-tolerance incurred by these schemes from the normal execution time of a process, we have also simulated a fault-free environment (referred to as the *base case*). In this environment, processes do not perform any message or file logging and output commit. Moreover, they directly exchange file data from their caches (rather than through the file server). The EXISTING, VERSIONING, and LFT schemes were evaluated in terms of the increase in the execution times of a message send or a file I/O instruction as compared to the base case.

To precisely define the evaluation metric, let N_{read}, N_{write}, N_{send} and $N_{receive}$, respectively, denote the number of Read, Write, Send, and Receive instructions executed by a process. Similarly, let E_{read}, E_{write},

E_{send}, and $E_{receive}$, respectively, denote the contribution of the Read, Write, Send, and Receive instructions to the total execution time of the process. Thus, if $P_f = P_{read} + P_{write}$ and $P_m = P_{send} + P_{receive}$, respectively, denote the probabilities of executing a file I/O or a message passing instruction in a program, the expected execution time of these instructions (collectively referred to as IO instructions) can be computed as:

$$\widehat{E}_{io} = \frac{1}{P_m + P_f}\left[P_{read} * \left(\frac{E_{read}}{N_{read}}\right) + P_{write} * \left(\frac{E_{write}}{N_{write}}\right) + P_{send} * \left(\frac{E_{send}}{N_{send}}\right) + P_{receive} * \left(\frac{E_{receive}}{N_{receive}}\right)\right]$$

Thus, if $\widehat{E}^b_{io}$ denotes the expected execution time of IO instructions in base case, then the overhead per IO instruction imposed by a fault-tolerance scheme can be defined as:

$$\widehat{\mathcal{O}} = \frac{\widehat{E}_{io} - \widehat{E}^b_{io}}{\widehat{E}^b_{io}}$$

Observe that the value of $\widehat{\mathcal{O}}$ is a function of the N, the number of processes in the system; P_c, P_m, and P_f, the fraction of computational, message passing, and file I/O instructions executed by a process; P_s, the probability that the file accessed by a process is being shared concurrently with other processes; B, the number of bytes read and written from the file server during each Read and Write operation; and the processor speed. Using our simulator, we are carrying out extensive set of experiments to characterize the dependence of $\widehat{\mathcal{O}}$ on these parameters, the results of some of our initial experiments are presented in the next section.

4.3 Experimental Results

The overhead imposed by a fault-tolerance scheme is dependent on the time required to write data onto stable storage (i.e., at a file server). To estimate this, we measured the time to perform Read and Write operations on our departmental NFS server. Since these measurements are dependent on the load at the server, we carried out two sets of experiments - one during a heavy load scenario (at 11 AM) and the other during a low load scenario (at 2 AM). The average time observed for reading and writing files of varying sizes for both of these scenarios are shown in Figure 1.

Assuming that the overhead of writing data to stable storage is equal to the time to read/write files to the NFS server under the low load scenario, we have measured the overhead imposed by each of the three fault-tolerance schemes using our simulator[3]. For all of these experiments, we have assumed that $P_c = 0.999999$. That is, one in a million instructions is either a message passing or a file I/O instruction (i.e., $P_m + P_f = 10^{-6}$) [21]. The ratio $\frac{P_m}{P_f}$ is varied from 0.05 to 20, to capture the range of applications in which the non-computational instructions are dominated by file I/O or message passing instructions. We have varied the number of processes N involved in a distributed application from 3 to 12; the number of bytes B read and written from the file server during each Read and Write operation from 4KB to 32KB; and processor speed from 100MIPS to 750MIPS. Finally, we have chosen $P_s = 0.3$ as a representative value for the degree of file sharing. The results of these experiments are described below.

[3]Note that since we have chosen the overhead of writing data to stable storage is equal to the time to read/write files to the NFS server under the low load scenario, the values of $\widehat{\mathcal{O}}$ presented in this section are likely to be smaller than what would be observed in practice.

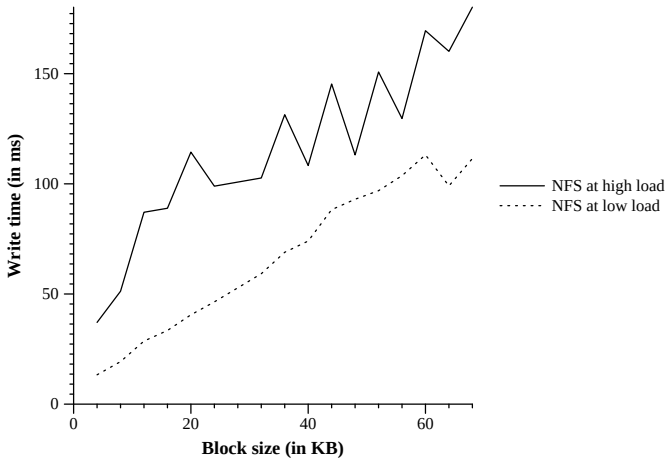

Figure 1 : Experimentally observed time required to write or transfer blocks of various sizes

4.3.1 Effect of Varying $\frac{P_m}{P_f}$ on $\widehat{\mathcal{O}}$

Figure 2 depicts the variation in $\widehat{\mathcal{O}}$ with increase in $\frac{P_m}{P_f}$ for a distributed application consisting of 6 cooperative processes. It demonstrates that: (1) as $\frac{P_m}{P_f}$ increases (i.e., as the fraction of file I/O operations decreases), the value of $\widehat{\mathcal{O}}$ decreases for all the fault-tolerance schemes; (2) the VERSIONING scheme incurs smaller overhead as compared to the EXISTING scheme; and (3) the overhead of LFT is negligible. A closer analysis of the results reveals that:

1. The overhead of logging and output commit incurred as a result of executing a Send or a Write is dependent on the amount of file information read between successive Send and/or Write instructions. As $\frac{P_m}{P_f}$ increases, the frequency of Read decreases, and hence, the cumulative overhead of output commit prior to a Send or a Write decreases.

2. Increasing the $\frac{P_m}{P_f}$ increases the frequency of Send/Receive instructions, and decreases the frequency of Read/Write instructions. When $\frac{P_m}{P_f}$ is increased starting from a very small value, the number of the number of Reads executed between a Send and a prior Send or a Write increases. Consequently, the logging overhead incurred by Send instructions increase. However, as the value of $\frac{P_m}{P_f}$ continues to increase, the frequency of file Read decreases, which, in turn, reduces the amount of information that needs to be logged on stable storage (and hence, decreases the overhead of output commit prior to a Send). Hence, the logging overhead for Send instructions first increases and then decreases with increase in $\frac{P_m}{P_f}$.

3. For a given value of P_s, the higher the frequency of file I/O operations (i.e., the smaller the value of $\frac{P_m}{P_f}$), the greater is the probability that the file being requested by a process is being held by some other process, and hence, greater is the overhead in obtaining the file data. Hence, the overhead of obtaining file data for a Read or a Write operation decreases with increase in $\frac{P_m}{P_f}$.

   For the VERSIONING scheme, the delay observed by a process p executing a Read or a Write is dependent on the probability that the asynchronous output commit initiated by process (say q)

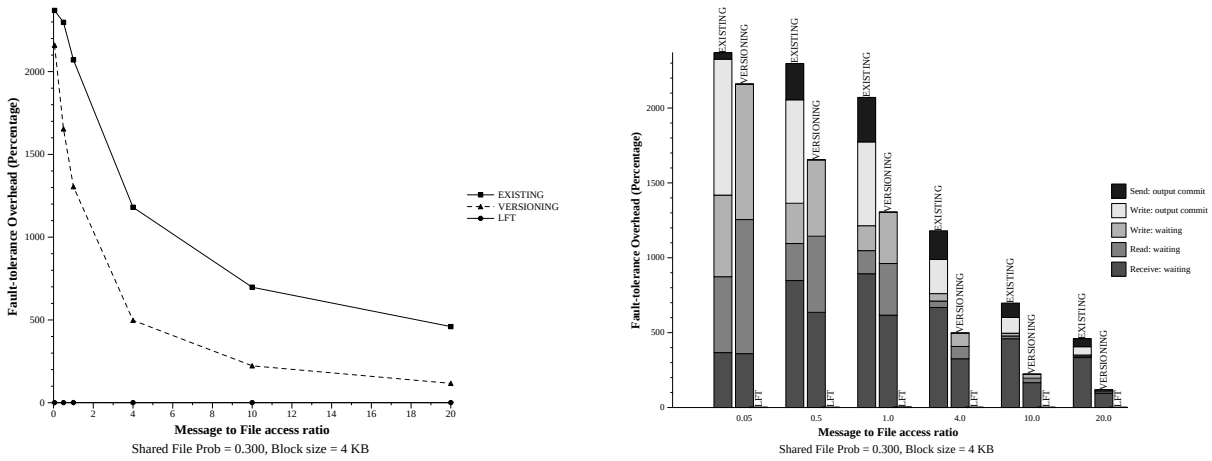

Figure 2 : Variation in $\widehat{\mathcal{O}}$ with increase in $\frac{P_m}{P_{i/o}}$ for $N = 6$, $B = 4KB$, and $P_s = 0.3$

currently holding the file has terminated prior to p's request. Hence, for a give value of P_s, the higher the frequency of file I/O operations, the greater is the probability that the asynchronous output commit initiated by process q has not completed when p issued a request for f. Hence, the waiting times for Read and Write operations are greater for the VERSIONING scheme as compared to the EXISTING scheme. Increase in the $\frac{P_m}{P_f}$ reduces the frequency of Read and Write operations, and thereby increases the probability that the asynchronous output commit initiated by process q has terminated prior to the access by p. Hence, the difference in the file Read and Write waiting times between the VERSIONING and EXISTING schemes reduce with increase in $\frac{P_m}{P_f}$.

4. For a fixed value of P_f, the higher the frequency of Receive operations (i.e., the higher is the value of P_m), the greater is the probability of requesting a message from a process that is blocked for file I/O. Similarly, for a fixed value of P_m, the lower is the the frequency of file I/O operations (i.e., the smaller is the value of P_f), the smaller is the probability of requesting a message from a process that is blocked for file I/O. The combined effect of these factors is that, with increase in $\frac{P_m}{P_f}$ (i.e., with simultaneous increase in P_m and decrease in P_f), the value of the message Receive overhead first increases and then decreases.

4.3.2 Varying Number of Processes

Figure 3 illustrates that increasing the number of processes in the system increases the value of $\widehat{\mathcal{O}}$ for the EXISTING and VERSIONING schemes, but has negligible impact on the performance of LFT. The increase in $\widehat{\mathcal{O}}$ for the EXISTING and VERSIONING schemes can be attributed to the following two reasons:

1. Increasing the number of processes in the system increases the number of accesses to the file server. For a given value of P_s, this increases the possibility of contention for shared files (i.e., the probability that a requested file is being held by some other process), and hence, increases file read and write waiting times.

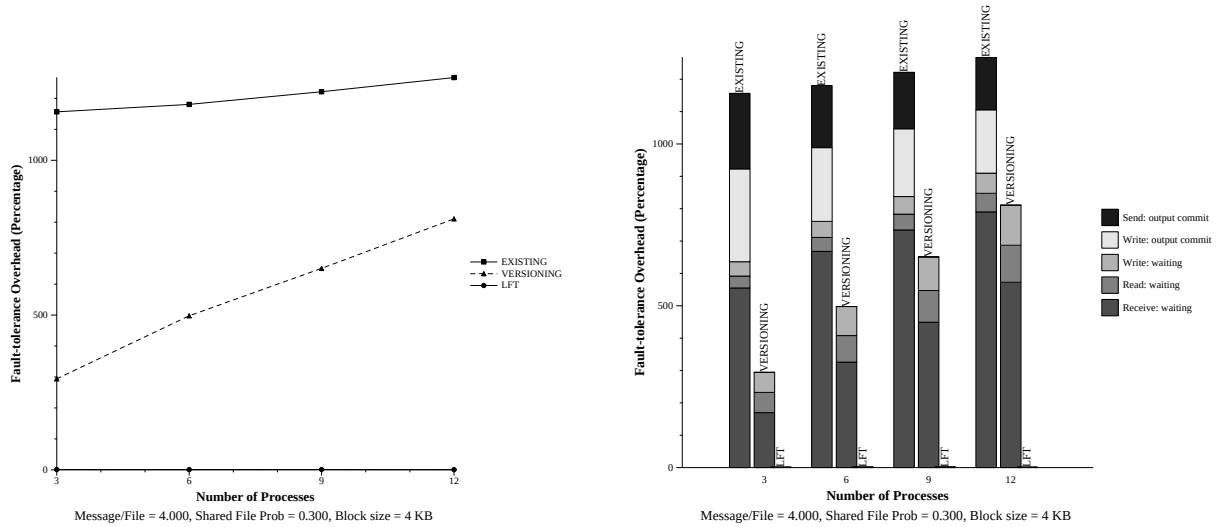

Figure 3 : Effect of variation in the number of processes on $\widehat{\mathcal{O}}$ for $P_s = 0.3$

2. Increasing the number of processes in the system increases the probability of creating long chains of Receive dependencies (i.e., process p may be waiting for a message from process q, which in turn may be waiting for a message from process r, and so on). As the number of processes in the system increase, the length of such a chain increases, thereby increasing the overhead of a Receive instruction.

4.3.3 Effect of Varying Processor Speed on $\widehat{\mathcal{O}}$

As processor speeds increase, the ratio of file I/O time to an instruction execution time increases. Consequently, the overhead $\widehat{\mathcal{O}}$ observed for the EXISTING and the VERSIONING schemes increase with increase in the processor speed. Once again, the impact of increased processor speed on the the overhead imposed by LFT is negligible. Figure 4 demonstrates this effect.

4.3.4 Effect of Varying File I/O Size on $\widehat{\mathcal{O}}$

For the EXISTING scheme, increasing the size of file I/O has two effects:

1. Increasing the amount of information read during a Read increases the amount of information that needs to be logged on stable storage prior to a Send or a Write. Hence, the overhead of output commit increases.

2. Since increase in the size of file I/O increases the overhead of Send, it indirectly increases the overhead of Receive.

Since the VERSIONING scheme eliminates the file logging required prior to Send instructions, and executes the output commit for Write instructions asynchronously, increasing the file I/O size has a much

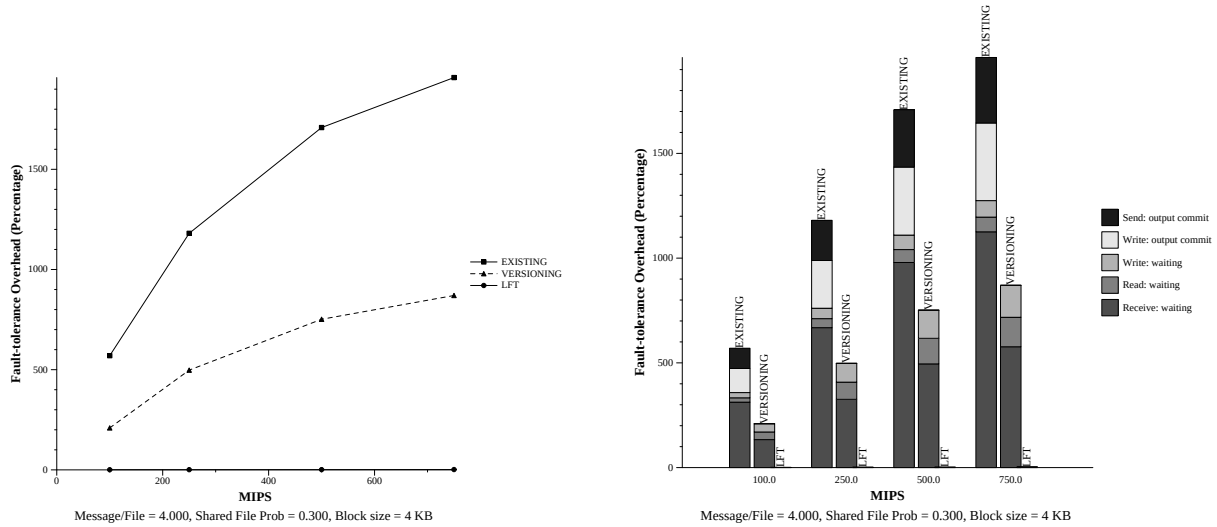

Figure 4 : Effect of variation in processor speed on $\widehat{\mathcal{O}}$

smaller effect on $\widehat{\mathcal{O}}$. The small increase in the overhead observed for the VERSIONING scheme can be attributed to the increase in file read and write waiting times.

Variation in the file I/O size has negligible effect on the value of $\widehat{\mathcal{O}}$ for LFT. Figure 5 depicts these effects.

5 Concluding Remarks

Recent advances in computing and communication technologies have enabled a new class of highly cooperative distributed applications. For many of these applications, explicit replication will be too costly to implement, or, simply, high availability will not be necessary. For these applications, the availability of low-overhead fault-tolerance techniques will be crucial to achieving reliability. To address these needs, we are developing *lightweight fault-tolerance* (LFT), a new low-overhead approach to fault-tolerance for highly cooperative distributed applications.

Many of the goals of LFT are shared by shared by other low-overhead fault-tolerance techniques. LFT, however, is unique in its focus on efficient support of fault-tolerance for applications in which communication occurs through both messages and files. Unlike most existing techniques which consider a file system as a detached component of the external environment, LFT presents the file system to the application as an integrated partner that can be trusted to provide the data needed during recovery.

In this paper, we described how LFT extends to file communication the causal logging techniques used in message passing. We demonstrated that in LFT, all the synchronous operations that are currently performed by log-based protocols during file I/O are either eliminated or made asynchronous. Through simulations, we demonstrated that LFT brings the cost of file communication down to the level of message passing, drastically reducing the overhead incurred by fault-tolerant applications in performing file I/O. In addition to achieving these performance gains, we argued that our approach has the potential to enhance the

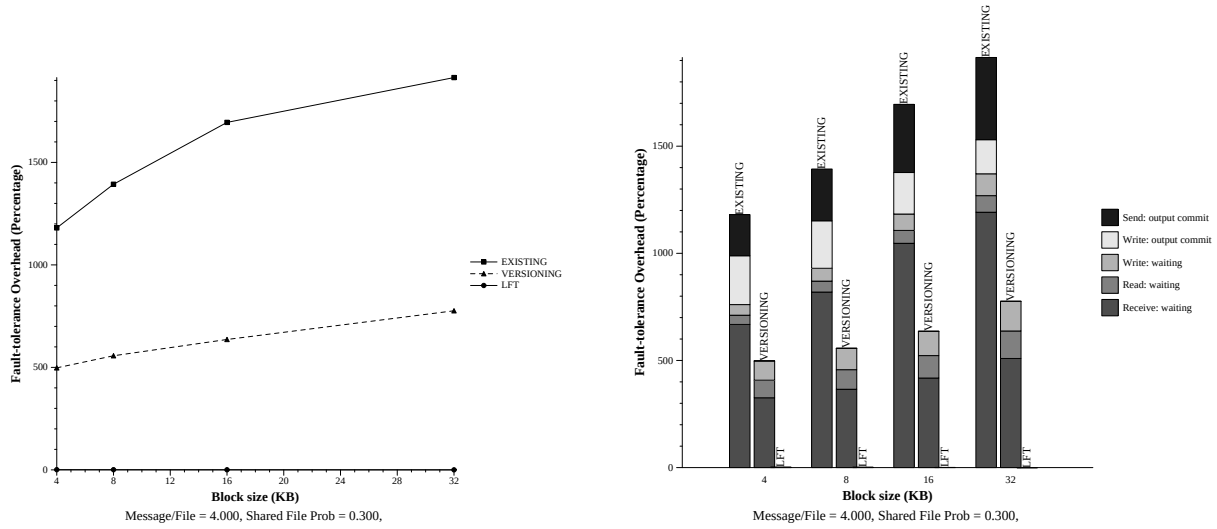

Figure 5 : Effect of variation in B on $\widehat{\mathcal{O}}$ for $N = 6$, $\frac{P_m}{P_f} = 4.0$, and $P_s = 0.3$

effectiveness of existing rollback recovery techniques for software fault-tolerance. LFT is currently being implemented as a middleware at the Department of Computer Sciences at UT Austin.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] Mary Louise Gray Baker. *Fast Crash Recovery in Distributed File Systems*. PhD thesis, University of California, Berkeley, 1994.

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

[7] Anita 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.

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

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

[10] 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.

[11] 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.

[12] J. Gray. Why do computer stop and what can be done about it? In *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, October 1993.

[13] 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.

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

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

[16] M.Kazar, B.Leverett, O.Anderson, V.Apostolides, V.Bottos, S.Chutani, C.Everhart, W.Mason, S.Tu, and E.Zayas. Decorum File System Architectural Overview. In *Proceedings of the Summer 1990 USENIX Conference*, pages 151–163, June 1990.

[17] J.C. Mogul. A Recovery Protocol for Spritely NFS. In *USENIX File Systems Workshop Proceedings*, pages 93–109, May 1992.

[18] J.C. Mogul. Recovery in Spritely NFS. Technical Report 93/2, Digital Western Research Laboratory Rerearch Report, June 1993.

[19] 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.

[20] 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.

[21] M. Rosenblum, E. Bugnion, S.A. Herrod, E. Witchel, and A. Gupta. The Impact of Architectural Trends on Operating Systems Performance. In *Proceedings of 15th Symposium on Operating Systems Principles*, pages 285–298, December 1995.

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

[23] 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.

[24] V. Srinivasan and J.C. Mogul. Spritely NFS: Experiments with Cache-Consistency Protocols. In *Proceedings of the 12th Symposium on Operating System Principles*, pages 45–57, December 1989.

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

[26] 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.

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

[28] B.B. Welch and J.K. Ousterhout. Pseudo-Devices: User-Level Extensions to the Sprite File System. In *Proceedings of the Summer 1988 USENIX Conference*, pages 37–49, June 1988.