Low-overhead Protocols for Fault-tolerant File Sharing

Lorenzo Alvisi Sriram Rao Harrick M. Vin
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188, USA

Abstract

In this paper, we quantify the adverse effect of file-sharing on the performance of reliable distributed applications. We demonstrate that file-sharing incurs significant overhead, which is likely to triple over the next five years. We present a novel approach that 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 agents 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 agents to exchange files directly, without first saving the files on disks at the server. As a consequence, the costs of supporting file-sharing and message passing in reliable distributed application become virtually identical. 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.

1 Introduction

Low-overhead rollback-recovery protocols—such as checkpointing and message logging [2, 3, 9, 17, 18]—have been extensively studied for message passing applications. These protocols seek to tolerate common failures while minimizing the use of additional resources and the impact on performance during failure-free executions. In this paper, we focus on low-overhead protocols for applications in which agents communicate both through message passing and file sharing. Our work is motivated by the qualitative observation that file sharing adversely affects the performance of today's reliable distributed applications. On the one hand, conventional file servers do not support file sharing efficiently: on receiving a file access request, they require the agent possessing the most recent version of the file to synchronously write-back the file at the server prior to servicing the request. On the other hand, conventional rollback-recovery protocols incur substantial overhead when used for applications in which agents communicate also through file sharing.

The first contribution of this paper is in quantifying the adverse effects of file sharing on performance of reliable distributed systems. We demonstrate 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. The second contribution of this paper is to present a protocol that virtually eliminates this overhead. The central idea of our solution is to track causal dependencies resulting from file sharing and to record them using *determinants*—tuples that identify file I/O and message passing operations and the order of their occurrence with respect to other events in an agent execution. We show that if determinants are available during recovery, then interactions with the file server can be reproduced, and file data lost in a failure can be regenerated. To ensure determinants availability, we use an efficient replication scheme [3] that stores determinants in agents' volatile memory. The final contribution of this paper is the introduction of 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 [4, 11]. 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.

The remainder of the paper is organized as follows. In Section 2, we describe our system model. The effect of file sharing on the performance of reliable distributed applications is quantified in Section 3. We describe our solution and its salient features in Sections 4 and 5, respectively, and then discuss the protocol implementation issues in Section 6. Section 7 discusses related work and finally, Section 8 summarizes our results.

2 System Model

We assume an asynchronous distributed system consisting of a set of agents and a file server. Agents communicate using both message passing and file sharing. Messages are exchanged over FIFO channels that can fail by transiently losing some messages. Files are shared according to an ownership-based consistency protocol [4, 11]. 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 $F.v$. Given a file F, a new version of F is created whenever F is modified. On accessing F, the file server returns the latest version of F.

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 [13]. For each agent 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. Deliver, read and write events are non-deterministic, because the order in which an agent receives messages and the file versions it accesses are execution-dependent. Send events and other local events are instead deterministic. Agent execution is *piecewise deterministic* [17]: It consists of a sequence of deterministic intervals of execution, joined by non-deterministic events. At any point during the execution, the *state* of an agent is a mapping of program variables and implicit variables (such as program counters) to their current values[1]. Given the initial state of each agent 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 agents 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 agents p and q are *mutually consistent* if they are both mutually message- and file-consistent. A collection of states, one from each agent, is a *consistent global state* if all pairs of states are mutually consistent [6]; otherwise it is *inconsistent*.

We assume that agents fail according to the fail-stop model [15]. The file server can fail independently and only by halting; however, its failure and recovery are not addressed in this paper. Finally, we assume that *stable storage* [10] is available throughout the system, persists across failures, and is implemented either using disks at the file server or through replication in the volatile memory of agents.

3 Problem Statement

The next generation of distributed applications will be structured around groups of agents that communicate in different ways. Tightly-coupled agents will use message passing — either directly or through distributed shared memory — to achieve low-latency; loosely-coupled agents, or agents that communicate without knowing each other's identity, will use file sharing.

Unfortunately, in today's distributed systems, file sharing adversely affects application performance. This can be attributed to the following two reasons. First, conventional file servers do not support file sharing efficiently. On receiving a file access request, they require the agent possessing the most recent version of the file to synchronously write-back the file at the server prior to servicing the request. Second, as the following example illustrates, conventional rollback-recovery protocols such as checkpointing and message logging [2, 3, 9, 12, 16, 17] incur substantial overhead when used for applications in which agents communicate also through file sharing.

Example Consider the execution in Figure 1, in which agents 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, existing message logging protocols proceed as follows: when an agent reads or writes a file, it blocks until the information necessary to prevent inconsistencies during recovery is logged to disk. On a read, the file read is saved on disk, to guarantee that it will be available during recovery. On a write, an *output commit* protocol writes to disk the data necessary to guarantee that the state in which the write is generated is recoverable. ■

This example illustrates the somewhat schizophrenic behavior that the fault-tolerance protocols developed for message passing applications exhibit when agents interact with the file server. These protocols rely on file servers to provide stable storage for checkpoints and other information used during recovery; yet when reading or writ-

[1] We assume that the state of the agent 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 agent.

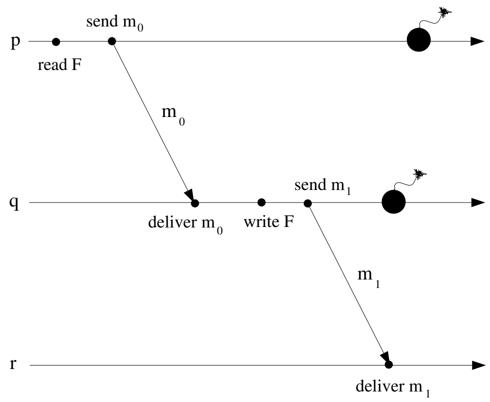

Figure 1: Example demonstrating the need for synchronous logging resulting from file sharing

ing other types of files, these protocols treat file servers as generic components of the external environment.

To quantify the overhead incurred by conventional file servers and fault-tolerance protocols for applications in which agents communicate also through file sharing, we simulated two environments Ω_1 and Ω_2. Ω_1 is an environment in which failures occur. To recover from them, agents use Family-Based Logging (FBL) — a roll-back recovery protocol described in Section 4.2. Ω_2 is a failure-free environment. The execution time of agents in Ω_1 differ from that in Ω_2 due to the following three factors:

1. *Synchronous write-back*: Due to file sharing, when an agent executes a read or a write event the most recent version of the file may be with another agent. In this case, Ω_1 blocks the agent executing the read/write event until the file server (1) obtains the most recent version of the file, (2) writes it to disk, and (3) transfers it to the requesting agent. In contrast, in Ω_2, on receiving a file read request, the file server informs the agent possessing the most recent version of the file to directly transfer the data to the requesting agent, without any synchronous write-back.

2. *Read logging*: Ω_1 blocks the agent executing a read event until the file data received from the server is logged to disk. In contrast, Ω_2 enables the agent executing the read event to resume execution immediately following the receipt of the file data.

3. *Output commit*: Ω_1 blocks the agent executing a write event until the completion of the corresponding output commit protocol. This involves writing to disk the order of messages delivered by the agent. Ω_2 imposes no such restrictions.

We modeled distributed applications as consisting of N agents, each executing a sequence of send, deliver, read, write, and other local events. The relative frequencies of these events were 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 an agent. Also, the extent of file sharing was controlled by P_s, which denotes the probability that the file accessed by an agent is being shared concurrently with other agents.

Using the simulator, we measured the execution time of agents in Ω_2 (denoted by E_{base}) and the increase in the execution time yielded by synchronous write-back, read logging, and output commit in Ω_1 (denoted by I_s, I_r, and I_o, respectively). Since the execution times of local, message send/deliver, and file read/write events depend on the processor speed as well as the latency and the bandwidth of disks and networks. We assume an environment with state-of-the-art workstations and networks (see Figure 3). Figure 2 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 2(a) and 2(b) demonstrate that increasing N and P_s increases the possibility that, when an agent executes a read or a write event, the most recent version of the file is with a different agent. 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 agents, 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 2(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 an agent 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 $P_f/P_m > 1$, an agent 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 2(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.

There are two important observations about the experiments just described.

1. In our simulations, we assume that Ω_1 blocks an agent 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 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 [12]. Figure 3 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 at the server significantly degrade application performance.

2. The simulations were seeded by processor, network, and disk performance values that represent the state-of-the-art workstations and networks. To put these results in perspective, we need to account for the expected technological trends [4]. Figure 3 illustrates that $\mathcal{O}$ will more than triple over the next five years.

From these experiments, we conclude that conventional file servers as well as fault-tolerance protocols developed for message passing applications incur substantial overhead for applications in which agents communicate also through file sharing. We believe that the main cause of this overhead is the *mutual mistrust* between file servers and these fault-tolerance protocols: the protocols do not trust file servers to provide during the recovery of a failed agent the files read by that agent prior to failure; and the file servers do not rely on the protocols' ability to recover failed agents to improve application performance. In what follows, we discuss the ideas that underlie our approach to eliminating this mistrust.

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

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

2. It allows recovering agents to regenerate files that were lost when these agents crashed. Thus, the file server can allow modified files to be shared among agents 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.

4.1 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 agent executing the non-deterministic event, (2) the relative order of this event with respect to other non-deterministic events executed by the same agent, 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 agent 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 agent executing e and the esn assigned to e by that agent.

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 [2]. Therefore, assuming that we can guarantee the availability of determinants (see Section 4.2), 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

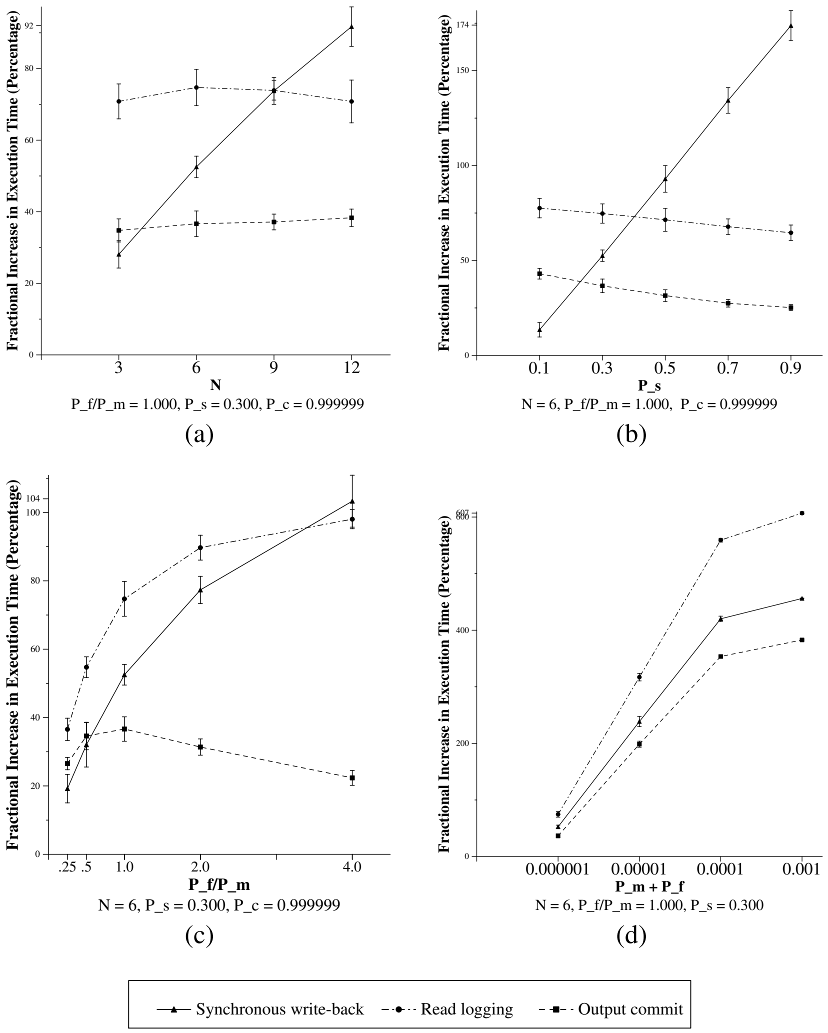

Figure 2: Variation in O_s, O_r, and O_o with N, P_c, P_f, P_m, and P_s

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

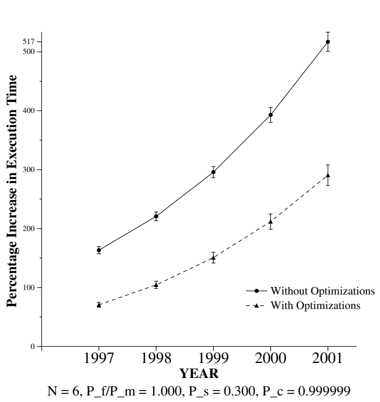

Figure 3: Effect of technological trends and optimizations on $\mathcal{O}$

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.

4.2 Logging Determinants Efficiently

Determinants are made *stable* by logging them on stable storage. This guarantees their availability during recovery. The performance of log-based protocols depends on the scheme used to make determinants stable. Our protocols are based on Family-Based Logging (FBL) — a logging technique we have proved to be optimal with respect to several significant performance metrics [2]. FBL was originally designed to log determinants of 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 the following, we briefly describe the main features of FBL and discuss how we generalize FBL to handle read and write determinants.

4.2.1 Family-Based Logging

FBL protocols are based on the following observation: in a system where agents fail independently and no more than t agents fail concurrently, stable storage can be implemented by replicating determinants in the volatile memory of $t+1$ agents. To specify FBL's logging technique, we introduce two sets for each non-deterministic event e:

Depend(e) This set includes the agent which executed e and any agent that executed an event e' such that e *happens-before* e' [13].

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

FBL guarantees that the following property holds:

$$\Box(\forall e : \neg stable(e) \quad : \quad (Depend(e) \subseteq Log(e)) \wedge \Diamond(Depend(e) = Log(e))) \quad (1)$$

where $\Box$ and $\Diamond$ are, respectively, the "always" and "eventually" temporal operators. 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 agent p causally depends on an event e, then either e is stable (*i.e.* either $|Log(e)| > t$ 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 $t+1$ times, if p causally depends on e, then $\#e$ is not lost.

4.2.2 Family-Based Logging for Efficient File Sharing

An agent 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 1. In fact, it is precisely because of the file server's inability to enforce this property that agents 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 agent. While in FBL agents piggyback determinants only on messages, now agents 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 agents. We improve on this scheme by observing that to satisfy Property 1 it is not necessary for determinants (or indeed even for files!—see Section 6.5) to be logged at the file server. It suffices that when an agent obtains a file, it receives with the file the non-stable determinants logged in the volatile memory of the file's last writer. Figure 4 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 agent that has the latest version of F (q in the figure); q then sends F to p, piggybacking on F all non-stable determinants in its volatile memory. The file transfer occurs directly between the agents, bypassing the file server that is used solely to locate the agent 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", agents have to contact the file server.

4.3 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 an agent has acquired ownership

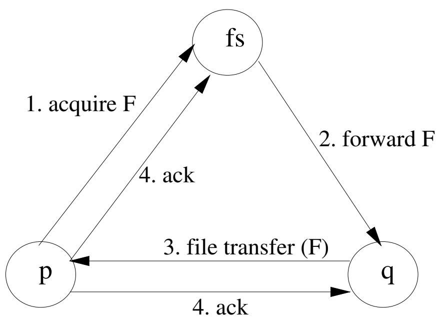

Figure 4: Example illustrating the file-sharing protocol

of a file version for reading or writing, then the outcomes of the subsequent read or write events executed by that agent are uniquely determined for as long as the agent 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 an agent acquires ownership of a file. A naîve implementation of this optimization, however, may complicate recovery. Consider, for instance, an execution in which an agent 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 [1].

We can reduce the number of file versions logged in the volatile memory of agents. Suppose an agent 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 agents in the system, p keeps only the last version of F in its volatile log.

5 Salient Features and Contributions

The proposed protocol has the following advantages:

- It eliminates synchronous output commits on writes. The protocol ensures that when an agent 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. Agents 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 an agent is stable. Our protocol only recovers the data necessary to restore the application to a consistent global state. For instance, if agent p creates $F.v$, sends $F.v$ to q, and both p and q crash before communicating with other agents, our protocol does not guarantee that $F.v$ will be regenerated during recovery, since the state of no correct agent 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 [14].

6 Protocol Design Issues

Due to space limitations, we do not present the pseudocode for our protocol. Instead, we identify the issues we faced in its design and present our solutions.

6.1 Notation

We represent file versions, application messages, 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 agent that created $F.v$, and $cesn$ is the event sequence number of the write event that created $F.v$

Application messages We represent an application message m with the tuple $\langle source, dest, sesn, data, piggyback \rangle$ where $source$ and $dest$ identify, respectively, the source and destination of m, $sesn$ is the event sequence number of the corresponding send event, $data$ is the content of m, and $piggyback$ stores the determinants that are piggybacked on m by $source$.

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 $F.v$.

6.2 Logging Determinants, Messages, and Files

Each agent 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 agents p knows to have logged a copy of $\#e$. p considers $\#e$ stable when either $\#e$ has been written to disk or $|dle.logged\_at| > t$.

Message Log : $MsgLog_p$ contains one entry for each application message sent by p. The entry mle in $MsgLog_p$ corresponding to a message m has the form $\langle sesn, dest, data, dv\rangle$, where the first three fields hold the same values of the corresponding fields in m, and dv is a vector of N event sequence numbers, called *dependency vector*. The $dv[q]$ counts the number of send, deliver, read and write events executed by q that causally precede the sending of m.

File Log : $FileLog_p$ contains an entry for each file version that p forwards to another agent. 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 dependency vector, and $destSet$ is a set consisting of the agents to whom p sent $F.v$.

To garbage collect these logs, the state of the application is periodically saved using a coordinated checkpoint.

6.3 Estimating $Stable(e)$

Since a determinant $\#e$ is piggybacked as long as it not believed to be stable, an agent which underestimates $|Log(e)|$ may needlessly piggyback stable determinants, making the messages on average significantly larger. To help agents keep their estimates accurate, different FBL protocols provide agents with different amount of information about the causal past of each non-deterministic event of which they log the determinant [3]. We describe here the simplest FBL protocol, in which an agent p estimates $|Log(e)|$ as follows:

1. If p generates $\#e$, it creates a corresponding entry dle in $DetLog_p$ and sets $dle.logged\_at$ to $\{p\}$.
2. If p receives $\#e$ for the first time, piggybacked on an application or file-transfer message sent by q, then p creates a corresponding entry dle in $DetLog_p$ and sets $dle.logged\_at$ to $\{p, q, \#e.dest\}$.
3. If p receives $\#e$ from q, and there exists already an entry dle in $DetLog_p$ for $\#e$, then p sets $dle.logged\_at$ to $dle.logged\_at \cup \{q\}$.
4. If p receives an acknowledgment for an application message it sent to q, then p (1) retrieves the entry dle in $DetLog_p$ that corresponds to $\#e$, (2) retrieves the entry mle in $MsgLog_p$ that corresponds to m, and (3) adds q to $dle.logged\_at$ if e is not stable and $dle.desn \leq mle.dv[dle.dest]$. Acknowledgments for file-transfer messages are treated similarly using $FileLog_p$ instead of $MsgLog_p$.

6.4 File Sharing

To achieve better performance, agents cache in their volatile memory those files for which they have acquired ownership. 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 $\langle fileId, cv, versionSet, copySet, mode\rangle$ 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 agents with a valid copy of $F.cv$ in their volatile memory, and $fte.mode$ records the mode (`READ` or `WRITE`) in which the agents in $fte.copySet$ have acquired $F.cv$.

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 agents 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 4). 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 $\langle p, \top\rangle$.

6.5 Failure Recovery

Recovery of an agent p proceeds in two phases.

Phase 1: In the first phase, p receives from each agent q two sets of determinants:

$DetSet_q$ 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 $dle.dest = p$.

$LogSet_q$ This set contains the determinants of events that q believes p had logged before failing. A determinant $\#e$ with a corresponding entry dle in $DetLog_q$ can enter $LogSet_q$ for one of the following reasons:

1. The event corresponding to $\#e$ was not executed by p, but q knows that p had logged a copy of $\#e$. That is, $dle.dest \neq p \wedge p \in dle.logged\_at$.
2. q piggybacked $\#e$ on an application message directed to p, which p did not acknowledge. To determine if $\#e$ was piggybacked on an application message m, q checks if it had logged $\#e$ before sending m. That is, if mle is the entry in $MsgLog_q$ corresponding to m, then $mle.dest = p$ and $dle.d\_esn \leq mle.dv[dle.dest]$.
3. q piggybacked $\#e$ on a file-transfer message directed to p, which p did not acknowledge. Once again, to determine if $\#e$ was piggybacked on a file-transfer message, q checks if it had logged $\#e$ before sending m. That is, if fle is the entry in $FileLog_q$ corresponding to m, then $q \in fle.destSet \wedge dle.d\_esn \leq fle.dv[dle.dest]$.

Furthermore, to reproduce the content of messages delivered and files accessed before crashing, p collects from each correct agent q the following two sets:

$MsgSet_q$ This set contains the application messages sent by q to p.

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

Finally, p obtains from each correct agent q the esn of the latest event of p that q depends upon:

$maxEsn_q$ This value is used in Phase 2 to help p determine when recovery has completed.

If other processes fail while p is recovering, then determining when Phase 1 is complete is a non-trivial problem [8]. To solve it, we use an algorithm that detects the end of Phase 1 without forcing correct processes to block while p recovers. A detailed description of our algorithm is beyond the scope of this paper.

At the end of Phase 1, p computes the following values:

$$DetLog_p = \cup_{q \in N} LogSet_q \qquad DetSet_p = \cup_{q \in N} DetSet_q$$
$$MsgSet_p = \cup_{q \in N} MsgSet_q \qquad FileSet_p = \cup_{q \in N} FileSet_q$$
$$maxEsn_p = \max\{maxEsn_q : q \in N\}$$

Phase 2: In the second phase, p re-executes the original run, using the determinants in $DetSet_p$ to replay correctly the non-deterministic deliver, read, and write events. The entries in $DetSet_p$ are matched with the corresponding entries in $MsgSet_p$ and $FileSet_p$ using esn_p, and agent p is rolled forward. Read and write events that have no corresponding determinants in $DetSet_p$ are performed on the cached version of the appropriate files. Recovery is complete when esn_p is equal to $maxEsn_p$. At the end of recovery, p uses the file server's $FileTable$ to determine which of the files in its cache are valid.

7 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 [14]; xFS implements cooperative caching to improve the performance of file reads [4]. 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 an agent prior to failure will be available during recovery. Similarly, implementation of stable storage, using special hardware, such as non-volatile RAM [5], or a specialized operating system, such as the Rio file cache [7], 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 agents 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.

8 Conclusion

File sharing adversely affects the performance of today's reliable distributed applications. 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 paper, 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 clients 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 agents 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.

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] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

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

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

[17] 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.

[18] 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.