Low-overhead Protocols for Fault-tolerant File-sharing

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

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.

Citation: Proceedings of International Conference on Distributed Computing and Systems (ICDCS'98), Amsterdam, Netherlands, May 1998 (*to appear*). Also available as Technical Report TR-98-01, Department of Computer Sciences, University of Texas, Austin, TX.

1 Introduction

Low-overhead fault-tolerance protocols—such as checkpointing and message logging [2, 3, 4, 13, 14, 17, 21, 27, 32, 30]—have been extensively studied for message passing distributed 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 such as checkpointing and message logging incur substantial overhead when used for applications in which agents communicate also through file sharing.

The first contribution of this paper is to quantify 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 continues 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 [2] 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. This implementation—a direct consequence of our ability to use replication in volatile memory to reproduce during recovery file data lost in a failure—drastically reduces the cost of file sharing. 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 [5, 16]. In our solution, no synchronous write is needed, and p can send F to q without delays.

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 messages. Files are shared according to an ownership-based consistency protocol [5, 16]. Specifically, the file server supports shared read-ownership, and exclusive write-ownership (i.e., a multiple-reader, single-writer policy). 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 [20]. 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* [27]: 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 [8]; otherwise it is *inconsistent*.

We assume that agents fail according to the fail-stop model [25]. That is, agents fail independently, only by halting, and a faulty agent is eventually detected by all correct agents. The file server can also fail independently and only by halting. However, its failure and recovery are not addressed in this paper. Finally, we assume that *stable storage* [15] 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, 4, 17, 19, 31] 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 $F.v_0$, p will access $F.v_1$. Because p reads a different version of F during recovery, p may not re-send m_0, or indeed any message, to q. This has two consequences. First, q may be unable during its recovery to re-send m_1 to r, leaving q and r mutually message-inconsistent. Second, q may be unable to reproduce the write event that generated the version $F.v_1$ read by p during recovery, leaving p and q mutually file-inconsistent.

To avoid such inconsistencies, existing message logging protocols adopt the following approach: when an agent reads or writes a file, it blocks until the information necessary to prevent inconsistencies during

[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.