Transparent Optimistic Rollback Recovery

David B. Johnson
Willy Zwaenepoel

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251-1892
dbj@rice.edu, willy@rice.edu

1 Introduction

Optimistic rollback recovery methods can efficiently and transparently provide fault tolerance for applications executing in a distributed system. With rollback recovery, information saved on stable storage during failure-free execution allows certain states of each process to be recovered after a failure. Examples of such methods include the use of *message logging and checkpointing* [12, 8, 3, 15, 14, 9, 13], and the use of *checkpointing* alone [11, 4, 2]. *Optimistic* methods in general allow unrecoverable states of one process to be seen by other processes, and optimistically assume that these states will become recoverable before a failure occurs. This allows the needed recovery information to be saved on stable storage asynchronously, reducing failure-free overhead. However, if after a failure, these states are not recoverable, processes other than those that failed may also be required to roll back in order to restore the system to a consistent state.

We have developed a theoretical model for reasoning about optimistic rollback recovery methods [9, 7], and have shown that, in any system using optimistic rollback recovery, there always exists a *unique* maximum recoverable system state. We have also developed two algorithms for finding this maximum recoverable system state. These results can be applied both to systems in which all execution of processes between received messages is assumed to be deterministic (e.g., message logging and checkpointing methods), and to systems in which no such assumption is made (e.g., checkpointing methods). We have completed a full implementation of optimistic message logging and checkpointing on a network on SUN workstations under the V-System, and performance measurements from it demonstrate the efficiency of this method [6]. The overhead on individual communication operations averaged only 10 percent, and the total overhead on distributed application programs ranged from a maximum of under 4 percent to much less than 1 percent.

This paper briefly describes the current status of our research. We also discuss some of its limitations and present a new algorithm that addresses these limitations [10]. This algorithm dynamically supports both deterministic and nondeterministic processes, and allows processes to individually switch between using message logging and checkpointing or using checkpointing alone.

This work was supported in part by the National Science Foundation under Grants CDA-8619893 and CCR-8716914, and by the Office of Naval Research under Contract ONR N00014-88-K-0140.

2 Current Status

Our model concisely captures the dependencies that exist within the system that result from communication between processes. The execution of each process is divided into a sequence of *state intervals*, such that in terms of the rest of the model, all individual states of a process within any single state interval are equivalent from the point of view of all other processes in the system. The differences between a deterministic and a nondeterministic process are limited to the respective definitions of process state intervals. Each state interval of a process is identified by a unique *state interval index*. A state interval is called *stable* if and only if some state of the process within that interval can be recreated from information on stable storage after a failure.

The current dependencies of a process are represented by a *dependency vector*, identifying the maximum index of any state interval of each other process on which this process depends. A system state is represented by a *dependency matrix*, composed of the dependency vector of some state interval of each process. A system state is *recoverable* if and only if it is *consistent* and each individual process state interval is *stable*. The process states that make up a system state need not all have existed at the same time. A system state is said to have *occurred* during some execution of the system if all component process states have each individually occurred. The *system history relation* defines a partial order on these system states, such that one system state precedes another if and only if it *must* have occurred first.

With this model, we have proven some important properties of any system using rollback recovery. First, the set of system states that have occurred during any single execution of a system, ordered by the system history relation, forms a lattice, called the *system history lattice*, with the sets of *consistent* and *recoverable* system states as sublattices. During execution, there is thus always a *unique* maximum recoverable system state, which never decreases. We have also proven sufficient conditions for committing output from the system to the "outside world," and for removing recovery information from stable storage when no longer needed.

We have developed two algorithms for determining this maximum recoverable system state, and have used the model to prove their correctness. The first algorithm finds the maximum recoverable system state "from scratch" each time it is invoked, whereas the second algorithm is incremental, beginning its search with the previously known maximum and utilizing information saved from its previous executions to shorten its search. We have completed an implementation of optimistic message logging and checkpointing using this first algorithm, running under the V-System [5]. Some of our performance measurements from this implementation, on a network of SUN-3/60 workstations, can be summarized as follows:

- The overhead on individual V-System communication operations averages only 10 percent, ranging from about 18 percent to 2 percent, for different operations.

- During a checkpoint, the execution of the process is suspended typically for only a few tens of milliseconds, since most data is written to the checkpoint before suspending the process. The total time to complete the checkpoint, though, is dominated by the time required to write the modified pages of the user address space to the checkpoint, and is is about 3 seconds per megabyte written.

- The total overhead experienced by distributed application programs is affected most by the amount of communication performed during execution. We measured the performance of distributed programs for solving the n-queens problem, the traveling salesman problem, and Gaussian elimination with partial pivoting. The total overhead ranged from a maximum of under 4 percent to much less than 1 percent.

- The time for recovery is dominated by the cost of restoring each failed process from its checkpoint, averaging about 1.5 seconds per megabyte of user address space. The running time of the algorithm to determine the maximum recoverable system state is negligible relative to the time required to restore the processes from their checkpoints and to replay the logged messages to the recovering processes.

To our knowledge, this is the only existing complete implementation of fault-tolerance using optimistic message logging and checkpointing.

3 Limitations

The lack of support for nondeterministic process execution is a significant limitation to current methods using message logging and checkpointing. Nondeterministic execution can arise, for example, through asynchronous scheduling of multiple threads accessing shared memory. To recover the state of a process using message logging and checkpointing, the sequence of messages originally received by the process after its checkpoint are replayed to it. The process is assumed to reexecute deterministically based on these messages, and to reach the same state as it had after receiving them before the failure. If process execution can be nondeterministic, recovery will not be successful. This limitation does not affect methods using checkpointing alone, since only process states recorded in checkpoints are used for recovery.

Another limitation of current message logging and checkpointing methods, which is shared by methods using checkpointing alone, is the difficulty of committing output from the system to the "outside world." Output must be delayed until the fault-tolerance support can guarantee that the sending process will never roll back beyond the state interval from which the output was sent. Essentially, this requires that all other state interval's on which this interval either directly or indirectly depends are recoverable after the failure. Without coordination between output and the message logging or checkpointing of individual processes, the delays in committing output may be substantial. These delays can be reduced by logging or checkpointing more frequently, but this can significantly increase the failure-free overhead of the system.

4 Future Directions

These observations lead us to a new algorithm called *output-driven optimistic message logging and checkpointing* [10], in which recording the needed recovery information on stable storage and determining the current maximum recoverable system state are both driven by the need to commit output from the system to the outside world. This algorithm allows all output to the outside world to be committed quickly after being sent, while reducing the overhead required to determine when such output can be committed. The algorithm further reduces fault-tolerance overhead by avoiding the logging of messages not needed to allow pending output to be committed. Each process commits its own state intervals as needed, and requires the cooperation of the minimum number of other processes. The algorithm is completely distributed with no centralized control.

The issue of nondeterministic execution is addressed by allowing individual processes to dynamically switch between using message logging and checkpointing or using checkpointing alone. We assume that processes can detect when their execution is nondeterministic, such as through a trap caused by the memory protection hardware. Processes can use message logging during deterministic execution, in order to avoid recording a new checkpoint each time they or other processes that depend on them need to commit output to the outside world. During nondeterministic execution, the process converts to using checkpointing alone. This feature can also be used by individual processes to reduce the overhead of message logging. Processes can decide not to log received messages during arbitrary periods of their own execution. This saves the overhead of copying each received message to a buffer in volatile memory, and the overhead of later writing this buffer to stable storage. Processes must then record a new checkpoint when they or other processes that depend on them need to commit output to the outside world. After each checkpoint, the process may begin logging messages again if desired.

We contend that there are significant advantages in allowing each process a dynamic choice between message logging and checkpointing and checkpointing alone. In particular, if one or more processes in the computation are nondeterministic, they would always use checkpointing, while other processes may choose to use message logging. Furthermore, if a process is known to be deterministic most of the time but occasionally experiences detectable nondeterministic events, it may choose to use message logging during deterministic periods, but turn off message logging after it has experienced a nondeterministic event. After a subsequent checkpoint, message logging may be turned on again until another nondeterministic event occurs. Deterministic processes can choose between message logging and checkpointing or checkpointing alone, depending on which they perceive to be the least expensive at any particular time. If a process receives a large number of messages during some period of time, it may choose to record a new checkpoint, eliminating the need for writing these messages to stable storage. If, on the other hand, a process receives few messages, but has a large and rapidly changing address space, it may instead decide to log these few messages and postpone taking an expensive checkpoint until it becomes necessary to do so for limiting the recovery time. This algorithm can be viewed as unifying the spectrum of methods between checkpointing alone and message logging and checkpointing.

We are currently implementing output-driven optimistic message logging and checkpointing in a system of diskless SUN workstations running the V-System [5] on an Ethernet network, using a shared network file server. Each node in the system runs a separate *recovery server* process, which executes the algorithm to commit process state intervals. The kernel records received messages in a buffer in memory until they are recorded on stable storage or discarded by the recovery server. The recovery server also manages recording checkpoints and restoring them during recovery. The algorithm allows each process a great deal of freedom in choosing between using message logging and checkpointing or using checkpointing alone. We are working on appropriate policies to enable each process to effectively make these choices to reduce its own fault-tolerance overhead.

We are also examining methods for exploiting limited application-specific knowledge to reduce the overhead of message logging, while still being transparent to the application. Our current approach to this is in the environment of a distributed shared memory system. In such a system, all messages between processes are generated by the shared memory system. Fault tolerance could be provided by simply logging these messages, but we believe it would be far more efficient to take advantage of the knowledge that these messages are sent to emulate a specific shared data structure. In a separate project, we are currently developing a distributed shared memory system using a variety of memory coherence mechanisms that are specific to the particular access pattern of each object [1]. As a simple example, if the shared memory system knows that a particular object is "read-only," accesses to it need not be logged. Wu and Fuchs [16] have recently proposed a pessimistic method for providing fault tolerance in a distributed shared virtual memory system, which in general requires processes to checkpoint on each interaction. We are interested in pursuing a more optimistic approach that reduces the number of checkpoints required.

References

[1] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 168–176. ACM, March 1990.

[2] Bharat Bhargava and Shy-Renn Lian. Independent checkpointing and concurrent rollback for recovery—An optimistic approach. In *Proceedings of the Seventh Symposium on Reliable Distributed Systems*, pages 3–12. IEEE Computer Society, October 1988.

[3] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[4] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[5] David R. Cheriton and Willy Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 129–140. ACM, October 1983.

[6] David B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* PhD thesis, Rice University, Houston, Texas, December 1989.

[7] David B. Johnson, Peter J. Keleher, and Willy Zwaenepoel. A simple algorithm for finding the maximum recoverable system state in optimistic rollback recovery methods. Technical Report Rice COMP TR90-125, Department of Computer Science, Rice University, Houston, Texas, July 1990.

[8] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *The Seventeenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 14–19. IEEE Computer Society, June 1987.

[9] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181. ACM, August 1988. To appear in *Journal of Algorithms*, September 1990.

[10] David B. Johnson and Willy Zwaenepoel. Output-driven distributed optimistic message logging and checkpointing. Technical Report Rice COMP TR90-118, Department of Computer Science, Rice University, Houston, Texas, May 1990.

[11] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[12] Michael L. Powell and David L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 100–109. ACM, October 1983.

[13] A. Prasad Sistla and Jennifer L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing.* ACM, August 1989.

[14] Robert E. Strom, David F. Bacon, and Shaula A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *The Eighteenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 44–49. IEEE Computer Society, June 1988.

[15] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[16] Kun-Lung Wu and W. Kent Fuchs. Recoverable distributed shared virtual memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.