Transparent Recovery in Distributed Systems
Position Paper

David F. Bacon

3 April 1990

1 Introduction

We are investigating transparent optimistic solutions to problems in distributed systems such as recovery [6], replication [3], parallelization [2], and concurrent competing alternatives [4]. By a *transparent* solution to such a problem we mean that a program is transformed automatically, and that the behavior of the program is equivalent to a possible behavior of the untransformed program; in addition, the programmer and the end-user need not be aware of the transformation.

Transparent solutions to such problems are relatively straightforward if synchronization is relied upon, but performance of such methods is generally poor or the implementation is too expensive, and they do not scale. Our approach is to use *optimistic* methods in which we guess that synchronization is unnecessary, and verify this asynchronously while the program continues execution. We track inter-process dependencies and log non-deterministic events so that we can roll back a computation that depends upon an incorrect guess. Where virtual memory virtualizes the space of a process, we virtualize time, "paging in" a previous process state when a "time fault" (or incorrect guess) occurs.

2 Optimistic Recovery

Our approach to recovery is based upon optimistic recovery [6] enhanced by optimizations to reduce the amount of logging [5, 1] and extensions which incorporate the filesystem and other external components into the recovery process [7, 8].

In optimistic recovery, we guess that processors do not fail; specifically, for every non-deterministic event (usually a message), we guess that there will not be a failure before that message has been asynchronously logged. Each of these guesses is assigned a number, and each process records the highest-numbered guess of every other process upon which it depends. In the event of a failure, the failed process is restarted and then synchronized with its neighbors by rolling back to a mutually consistent state. Thus recovery is more expensive than in a conservative mechanism, but failure-free performance is substantially improved because checkpointing, logging, and much of committing can be done asynchronously and concurrently with the normal execution of the program.

3 Transactions Are Insufficient

It is our position that transparent fault-tolerance is required because transaction-based systems are insufficient for large distributed applications. There are two primary reasons for this.

Firstly, transactions only recover data, not process state. This was acceptable in the centralized computing environment in which transactions were developed, but in a large distributed system, the *state* of failed components must be restored for there to be true fault-tolerance. For instance, in a distributed application consisting of a collection of display, compute, and database servers, if one database node crashes and restarts, it must somehow re-establish its connections with the other components of the system and agree on what work has been performed, and only then can the distribute application continue execution. Transactions provide no support for this problem of reconstructing a consistent state for a distributed application. As a result, unless all of this synchronization and agreement has been programmed explicitly (and correctly), the application will fail even though the database has not.

Because of this, writing truly fault-tolerant distributed application would require extensive additional coding for state-recreation and inter-process synchronization. This code will be complex and less likely to be well tested, since it is not in the main path of the application and the number of failure modes will be large. As a result, the code which is supposed to provide fault-tolerance will be the least reliable component of the entire system. On the other hand, this code will most often simply be omitted since a very large proportion of software written simply aborts the entire application when an

unexpected error is encountered.

The second problem with transactions is that even when only data recovery is required, the programmer must program for failure explicitly by (1) ensuring that transactions are sufficiently short, and (2) explicitly handling aborts by informing the user, retrying the transaction, and so on. If either of these issues are not properly addressed, the application will not be fault-tolerant.

4 Transparent Solutions and Their Limitations

Optimistic recovery solves these problems. As a result, programs can be much simpler: there is no need to structure the application as a collection of transactions, and all of the error recovery to recreate a consistent state is part of the underlying recovery mechanism. Since the error recovery is part of the system, it is more likely to be correct. Since the program itself is smaller, it is more likely to be correct as well. It is our belief that by supporting this kind of transformation in the underlying system, software will be made easier to write and less prone to failure.

However, just as there are still a few applications for which it is necessary to write a custom pager, optimistic recovery will not be able to support all applications.

Optimistic recovery also does not perform atomic updates over multiple sites; however, since optimistic recovery is not doing concurrency control, atomicity is not generally an issue. In addition, optimistic recovery does not provide for a user-initiated abort, since there are no transactions to abort in the first place.

We are investigating issues in concurrency control and how the problems of recovery and concurrency control can be solved independently in such a manner that either or both could be incorporated as needed [9].

Since all of our work is based on tracking causal dependencies, we can use a uniform set of commit guards to track various predicates. This allows us to abort computations resulting from processor failure, concurrency control conflicts, or other conditions by simply rolling back to a mutually consistent system state in which none of our conditions are violated. This will greatly simplify the problem of incorporating multiple transformations. We are continuing to study the problems of recovery, concurrency control, replication, and concurrent competing alternatives to work on a system in which all of these transformations can be applied together as required by

particular applications.

References

[1] BACON, D. F. How to log all filesystem operations (while only writing a few to disk). Research Note RC, IBM T.J. Watson Research Center, 1990.

[2] BACON, D. F., AND STROM, R. E. Optimistic parallelization of communicating sequential processes. Research Note RC, IBM T.J. Watson Research Center, 1990.

[3] GOLDBERG, A. P., AND JEFFERSON, D. Transparent process cloning: A tool for load management of distributed systems. In *Proceedings of 1987 International Conference on Parallel Processing* (August 1987), pp. 728 – 734.

[4] SMITH, J. M. *Concurrent Execution of Mutually Exclusive Alternatives.* PhD thesis, Columbia University, 1989.

[5] STROM, R. E., BACON, D. F., AND YEMINI, S. A. Volatile logging in n-fault-tolerant distributed systems. In *The Eighteenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers* (June 1988), pp. 44–49.

[6] STROM, R. E., AND YEMINI, S. A. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems 3*, 3 (August 1985), 204–226.

[7] STROM, R. E., YEMINI, S. A., AND BACON, D. F. Toward self-recovering operating systems. In *The International Conference on Parallel Processing* (1987), North-Holland.

[8] STROM, R. E., YEMINI, S. A., AND BACON, D. F. A recoverable object store. In *Hawaii International Conference on System Sciences* (1988), IEEE CS.

[9] YEMINI, S. A., STROM, R. E., AND BACON, D. F. Improving distributed protocols by decoupling recovery from concurrency control. Research Note RC 13326, IBM T.J. Watson Research Center, 1987.