Asynchronous Checkpointing for PVM Requires Message-Logging

Kevin Skadron

18 April 1994*

Abstract

Distributed computing using networked workstations offers cost-efficient parallel computing, but the higher rate of failure requires effective fault-tolerance. Asynchronous consistent checkpointing offers a low-overhead solution.

Parallel Virtual Machine (PVM) allows a heterogeneous network of UNIX workstations to serve immmediately as a distributed computer by providing message-passing services implemented on top of UNIX inter-process communication.

We briefly show that correct user-level support for an aggressive, asynchronous two-phase-commit checkpointing protocol for PVM's virtual circuit mode requires message logging.

1 Introduction

Networks of workstations are readily accessible, and an environment providing the necessary support for message-passing or shared-memory can immediately turn such a network into a distributed computer. UNIX, being portable and the dominant operating system in research and high-performance computing, is frequently the base for such environments. While a distributed computer like this may not be as efficient as a network of workstations built directly for the purpose, using an existing network and operating system is inexpensive and convenient. The higher rate of failure in such systems, however, requires effective fault-tolerance.

Parallel Virtual Machine (PVM) [1] is the most common UNIX-based environment for distributed computing. It implements a message-passing environment for a heterogeneous network of computers. It provides two modes: direct process-to-process communication using UNIX sockets (virtual circuits), or daemon-to-daemon communication, where the message is first passed to the PVM daemon on the sender's machine, then to the daemon on the recipient's machine, and finally to the recipient. PVM lacks an efficient mechanism for fault-tolerance; *Fail-safe PVM* [9] uses a slower sync-and-stop protocol.

We set out to implement for PVM's virtual circuit mode the aggressive two-phase-commit checkpointing protocol described by Elnozahy, Johnson, and Zwaenepoel [5], in which computation is only suspended long enough to fork a thread; checkpoints are written concurrently with computation. The overlap permits very low failure-free overhead, typically less than 5% even with frequent checkpoints. We find, however, that a correct user-level implementation for PVM requires message logging. UNIX sockets hide acknowledgements in the kernel, but the protocol requires control of acknowledgements in the message routines. Kernel modification is not an option; one of PVM's chief virtues is that it is strictly user-level code and can be installed by any user on almost any system.

This paper describes the need for message logging in implementing the Elnozahy protocol for PVM. Section 2 gives a brief overview of the semantics of consistent checkpointing. Section 3 describes the need for logging. Section 4 describes related work, and we summarize our work in Section 5. Our study was done using PVM version 2.4 [1].

2 Semantics of Consistent Checkpointing

The system is assumed to consist of a number of fail-stop processes [10] comprising a distributed application, with one or more processes at each processor node. Processes are assumed to communicate only by passing

*revised 7 February 1996; was "Modifications to PVM for Distributed Checkpointing under UNIX"

messages. A *process checkpoint* consists of saving that process's state (memory contents and process context). An *application checkpoint* consists of one checkpoint each per process comprising the application. This set of checkpoints must represent a globally consistent state: the state that would exist upon restarting the application from its checkpoints must not be a state impossible during uninterrupted operation. For example, a state in which some process has received a message which no process has sent is not allowed.

When a failure is detected, the application terminates and restarts from its checkpoints, with the faulty node not used. Checkpoints must be stored on a central server to provide maximum availability of the checkpoints; storing checkpoints on individual nodes makes their availability subject to the node failures which are to be defended against. In the case of uninterrupted server operation (a reasonably reliable server can be attained through replication), consistent checkpointing is resilient to an arbitrary number of failures.

The individual processes can only affect each other through interprocess communication [3]. This means efficient fault-tolerance can be achieved by allowing processes to continue computing while they checkpoint. Synchronizing communication is sufficient to achieve consistency. The two disallowed states are a message which has been received but not sent (see Figure 1), and a message which has been successfully sent but not received (see Figure 2).

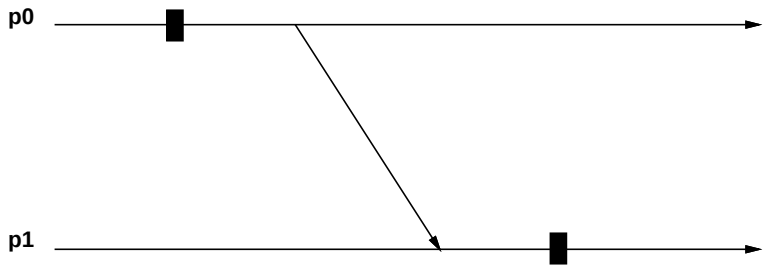

Figure 1 *p0 has recorded its checkpoint before sending the message; if restarted from the checkpoint, it will restart in a state in which the message remains to be sent. p1, however, receives the message before checkpointing, and will restart in a state in which it has received a message that appears never to have been sent.*

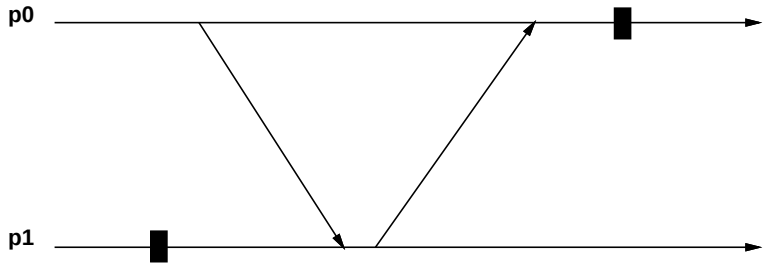

Figure 2 *p1 has recorded its checkpoint before receiving the message; if restarted from the checkpoint, it will restart in a state in which the message has not yet occurred. p1, however, sends the message and receives the acknowledgement before checkpointing, and will restart in a state in which it has already successfully sent the message and will not resend it.*

Consistency is achieved using an asynchronous two-phase-commit protocol that restricts a process's view of messages while checkpointing, but otherwise allows computation to continue. Two-phase-commit ensures that the application's checkpoints all see a message as either successfully sent and received, or not successfully sent and not received. This is accomplished by tagging each message with a *consistent checkpoint number* (CCN) corresponding to the sender's current checkpointing epoch. A process must defer receipt of any message or acknowledgement tagged with a CCN higher than its own until it too reaches that epoch.

In the first phase a distinguished checkpoint *coordinator* initiates a checkpoint by sending to each process a *trigger message*.[1] On receipt of a trigger, a process increments its CCN and takes the checkpoint. In the second phase, each process replies with a *success message* if its checkpoint succeeded. When success messages have been received from all processes, the coordinator sends to each a *commit message*, which causes the tentative checkpoint to be made permanent and the previous permanent checkpoint to be discarded. If there is a failure, an *abort message* is sent instead, the previous checkpoint is maintained, and the tentative checkpoint is discarded; failure recovery is initiated if deemed necessary.

Use of the CCN ensures that all processes agree on the state of messages within an application checkpoint. If some process sends a message in its epoch $N+1$, all recipients will defer receipt of that message until they reach their epochs $N+1$. If some process sends a message in its epoch N and the recipient acknowledges in its epoch $N+1$, the recipient's checkpoint N has not seen the message, because the message hasn't officially been received until the acknowledgement can be sent; the acknowledgment is deferred by the sender until the sender also enters epoch $N+1$, and the sender's checkpoint N sees the recipient as not having acknowledged, interpreting this on restart as a failed send. In both scenarios, on restart all processes agree the message "never happened".

3 Implementing Consistent Checkpointing for PVM

3.1 The Problem

Implementing asynchronous two-phase-commit requires access to the message facilities in order to append and check CCNs. The difficulty is that PVM's virtual circuit mode uses *sockets*, which are implemented using TCP/IP in the UNIX kernel. Sockets present access to streamed communication, which appears to the user as a two-way continuous stream of bytes (hence "virtual circuit"); the sender can write any number of bytes and the receiver can read any number of bytes. Reliability is built in; acknowledgements, resending, and duplicate elimination are all handled transparently by the kernel.

PVM's messaging routines, which are user-level code, thus cannot modify socket behavior to append CCNs to messages. CCNs can easily be added to PVM's own message header, and this allows a recipient to determine when a message should be deferred. This is not sufficient. The kernel performs acknowledgements, but CCNs need to be appended to them, too. When doing a send, as soon as the message arrives at the destination machine, that kernel sends an acknowledgement back and the sender regards the message as successfully received. This creates the potential for sent-but-not-received inconsistencies.

3.2 The Solution

The best solution is sender-based logging. The sender simply saves messages, and in the event of a restart after a failure, all processes must go through a special synchronization phase in which each process announces the last message it has seen, and any messages which have been lost by that process are retransmitted. This in fact amounts to allowing the messaging routines to control acknowledgements. Garbage collection is easily achieved by exchanging vectors containing last-seen values, and this can take advantage of the fact that processes must already perform periodic synchronization in order to checkpoint.

Sender-based logging only requires minor modifications to the described protocol, and because the buffer of sent messages is saved with the checkpoint, this scheme remains resilient to an arbitrary number of failures. Additional overhead during failure-free operation consists only of saving and managing the messages.

4 Related Work

Koo and Toueg [8] describe consistent checkpointing using a two-phase-commit protocol, with a focus on minimizing the number of processes which must take part in a checkpoint. They prove the need for a resilient checkpoint algorithm to store at least two stable checkpoints.

[1] The coordinator uses some form of timer to initiate the checkpoints at regular intervals.

Elnozahy, Johnson, and Zwaenepoel [5], whose two-phase-commit protocol we study, demonstrate excellent performance for consistent checkpointing under the V-System. This system exhibits less than 1% overhead for a wide range of compute-intensive applications representing a range of memory and communication behavior, and at most 6% overhead for any application tested. These figures are for checkpoints taken every 2 minutes. Their implementation relies on kernel support; this also facilitates incremental checkpointing, in which only the state which has changed since the last checkpoint is written to disk. While a user-level implementation such as PVM with checkpointing cannot hope to achieve comparable performance, it has the advantage of platform-independence among UNIX systems, like PVM itself.

León, Fisher, and Steenkiste [9] have implemented *Fail-safe PVM*, which takes coordinated checkpoints by issuing a global barrier and waiting for a quiescent state, thus forcing a consistent state. This sync-and-stop protocol, however, imposes a heavy penalty on failure-free performance. Our method of asynchronous checkpointing using a two-phase-commit protocol avoids such delays; the only sources of overhead are contention due to coordination, contention at the disk from writing of the checkpoint, and managing the sender logs.

Borg *et al* [2] describe an implementation of UNIX, TARGON/32, which provides completely transparent fault-tolerance, but the system relies on the existence of atomic three-way message delivery. TARGON/32 provides fault-tolerance for distributed applications by syncing each primary process with an inactive backup process on a different node. It requires all processes wishing to take advantage of this service to only communicate via supplied messaging routines. Reported overhead is 10%. Recovery is automatic if the failure does not also involve failure at the node with a needed backup process.

Elnozahy and Zwaenepoel [6] analyze the performance of consistent checkpointing, message-logging, and Manetho (a hybrid of uncoordinated checkpoints and message logging) [7], and conclude that for applications where output latency is not critical, consistent checkpointing is superior to the other methods.

5 Conclusions

PVM provides a convenient and inexpensive way to turn a network of workstations into a distributed computer, but lacks an efficient means of fault-tolerance. We describe an aggressive, asynchronous two-phase-commit protocol for consistent checkpointing, originally suggested by Elnozahy *et al* [5]. For correct user-level implementation under PVM, this protocol requires sender-based logging. Such a system will provide the superior performance of asynchronous consistent checkpointing for a package which can be installed by a user on any UNIX system.

6 Acknowledgements

This work was supported by the Rice Undergraduate Scholars Program, a program encouraging undergraduate research and operated by the Rice University School of Natural Sciences. Additional support was provided by the Rice University Department of Computer Science.

The author would like to thank Dr. Willy Zwaenepoel of Rice's Department of Computer Science for his support and guidance; and Marina Drobnic and Larry Weiner of Rice's Department of Electrical and Computer Engineering, who did heroic work in implementing a prototype of checkpointing for PVM [4].

References

[1] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A users' guide to PVM Parallel Virtual Machine. Oak Ridge National Laboratory, ORNL/TM-11826, September 1992.

[2] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1): 1-24, February 1989.

[3] 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.

[4] M. Drobnic, L. D. Weiner, and K. Skadron. Atlas: consistent checkpointing under UNIX. Final class presentation, *Computer Science 520*, Rice University, December 1993.

[5] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the Eleventh Symposium on Reliable Distributed Systems*, October 1992.

[6] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. To be presented in *Proceedings of the 1994 Fault Tolerant Computing Systems Symposium*, June 1994.

[7] 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

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

[9] J. León, A. L. Fisher, and P. Steenkiste. Fail-safe PVM: a portable package for distributed programming with transparent recovery. *CMU-CS-93-124*. February, 1993.

[10] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222-238, August 1983.