Rollback Recovery of PVM Applications *

Denis Conan, Pierre Taponot and Guy Bernard

Institut National des Télécommunications
9 rue Charles Fourier 91011 EVRY Cedex France
Fax: +33–1–60 76 47 80
e-mail: conan@int-evry.fr

Abstract

PVM: Parallel Virtual Machine [Geist94] is a software system that permits a heterogeneous collection of Unix computers networked together to be viewed by a user's program as a single parallel computer. PVM uses the message passing model to allow programmers to exploit distributed computing. Unfortunately, PVM does not tolerate computer crashes.

In this paper, we show how rollback recovery can be simply added to PVM in order that PVM applications survive computer crashes. For this, a mechanism developed at the I.N.T. is used [Bernard94]. The fault tolerance software checkpoints all tasks, logs interprocess messages and recovers the virtual machine by rollbacking a minimum of processes.

We obtain a solution that is simple thanks to a new feature introduced in the version 3.3 of PVM: the scheduler also called the resource manager. The solution is very efficient because the fault tolerance software permits asynchronous checkpointing and message logging and only minor modifications are inserted in the source code of PVM daemons.

Keywords

Distributed system, distributed computing, PVM, fault tolerance, rollback recovery.

*Accepted to the Romanian Open Systems Event (ROSE '95), Bucharest, Romania, 1-4 November 1995.

1 Introduction

Networks of workstations become more and more widespread computing environments, in particular because of their attractive performance/cost ratio. In such an environment, migrating from centralized applications (executing on one machine) to distributed applications (executing simultaneously on several machines of the network) is a natural evolution. However, several problems arise due to the distribution. Among them are the management by programmers of numerous distributed units and the sensitivity of distributed applications to machine failures. The PVM: Parallel Virtual Machine [Geist94] is an already famous answer to the first problem just cited and rollback recovery [Strom85, Borg89, Johnson90, Elnozahy92, Conan, Sens95] implementing passive replication is the cheapest solution ever found to the second one.

In the paper, we consider a network of workstations running a Unix operating system. Our computation model of a distributed application consists in several processes running in parallel on different machines, with communication links between them. PVM makes a collection of computers appear as one large virtual machine. It transparently handles all message routing and task scheduling. With the programming interface, the user writes his/her application as a collection of cooperating tasks. Tasks access PVM resources through a library of interface routines. These routines allow the initiation and the termination of tasks as well as communication and synchronization between tasks. Communication and synchronization constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast and barrier synchronization.

PVM introduces two kinds of processes: daemons and tasks. One PVM daemon noted *pvmd* runs on each computer of the virtual machine. The pvmd serves as a message router and controller. The first pvmd (started by hand) is designated the *master*, while the others (started by the master) are called *slaves*. User processes started by the master or by a slave are registered as tasks in the virtual machine. More complex tasks were introduced in version 3.3: for instance, *resource managers* also called *schedulers*. A scheduler is a PVM task responsible for making task and daemon placement decisions. There are already simple schedulers embedded in pvmds. For the sake of simplicity, complex tasks other than the schedulers are not considered in this study.

Several methods can be used to provide fault tolerance in distributed systems. Using a specialized hardware may be efficient, but such a component cannot be easily added to existing systems. Application-specific methods and atomic transactions require the use of a particular programming model (perhaps incompatible with the message passing paradigm). Active replication is well suited for real time systems but requires the use of extra processors.

We propose to handle machine failures using checkpointing, message logging and rollback-recovery. When a machine failure is detected, a replacement machine is found in the network. Practically, a checkpointing algorithm registers from time to time the state of the distributed application and a recovery algorithm restarts the distributed application from a previous state in case of a machine failure. To our knowledge, only one technical report presents a fault tolerant PVM: it is the work of Leon *et al.* at Carnegie Mellon University [León93]. In their solution, almost every entity of the virtual machine must rollback in case of a single machine failure. Here, we propose a better solution built from the new concept of the scheduler. For a first study, we don't treat the failure of the machine where the master and the scheduler run.

This paper is organized as follows. Section 2 demonstrates that the scheduler is necessary to adding rollback recovery to PVM. Section 3 develops scheduler's actions during the recovery. Section 4 details modifications inserted in the PVM's source code. Section 5 compares the proposed solution with related works. Section 6 concludes and gives the direction of ongoing works at the I.N.T.

2 Why it's necessary to add a scheduler to PVM

First, we have to determine precisely which entities compose the distributed application that the fault tolerance software has to take into account. A first naive answer will be all the virtual machine - i.e. pvmds and user's tasks. But, rollback recovery using message logging only supports deterministic processes in the sense that, if two processes start in the same state, execute the same portion of code and both receive the identical sequence of inputs, they will produce the identical sequence of outputs and will finish in the same state. In PVM, pvmds communicate with one another through UDP sockets and because UDP is an unreliable delivery service which can lose, duplicate or change the order of delivery of packets, an acknowledgment and retry mechanism is used. So, the latter uses timers and makes the execution of pvmds not deterministic. Therefore, the fault tolerance software cannot support pvmds, i.e. every pvmd belongs to the outside world of the fault tolerant distributed application and is not fault tolerant.

The question is then which user's entities could manage PVM resources after the crash of the master - i.e. which fault-free entities manage the virtual machine's configuration. The configuration depends on the initial configuration described in user's *hostfile* configuration file and that the user can also change the configuration interactively. Thus, the configuration has to be maintained by user's tasks. These tasks are the schedulers. Note that the schedulers are included in the set of fault tolerant processes.

So, we program a generic - i.e., application independent - scheduler. For the sake of simplicity, we limit the number of active schedulers in the virtual machine to one and thus don't allow more than one process to register as a scheduler. Practically, the user starts the master and immediately after the scheduler on the same computer. As a matter of course, the scheduler running on the master host manages any slave pvmds that don't have their own schedulers, i.e. all the virtual machine.

The scheduler consists in an infinite loop that successively accepts a request and then changes the configuration of the virtual machine. Every user's request is automatically redirected towards the scheduler. It is then the responsibility of the scheduler to treat the request's arguments, call the right PVM routines and control the evolution of the virtual machine.

3 The scheduler during the recovery

During fault-free execution, the fault tolerance software transparently checkpoints the distributed application and logs inter-task messages. During the recovery, the scheduler is responsible for managing the configuration of the virtual machine. Thus, the scheduler and the fault tolerance software interact. Here, we detail the scheduler's actions during the recovery. For more information on the fault tolerance software, see [Bernard94, Conan95].

The first stage of the recovery is the detection of crashes. When a machine crashes, the daemon running on this host is down. Another pvmd trying to communicate with the first will detect the failure through his triggered timeouts. When a slave pvmd has detected a shutdown, it takes two actions: inform the master of the failure and erase all the messages in transit towards the failed host. When the master has detected a shutdown, it informs all the slaves and the scheduler of the failure and erases all the messages in transit towards the failed host. Moreover, each pvmd sends a keep-alive message to remote pvmds once in a while. So, the failure is detected in finite time.

Because the fault tolerance is transparent to the user, the *"notification"* messages provided by PVM as means to implement fault recovery in an application are not sent to the tasks.

Following the detection of the failure, the fault tolerance software must stop the sending of interprocess messages for all the entities of the distributed application. The objective is to avoid in-transit messages that will become orphan after rollbacking *[Conan, Conan95]. As there is no explicit coordination be-

*The fault tolerance software doesn't avoid orphan messages but minimizes their number and rollbacks orphan processes due to orphan messages.

tween PVM and the fault tolerance software and in order to minimize the number of orphan messages, pvmds now stop inter-task messages too. The master sends a *"stop"* message to every pvmd and each pvmd stops reading tasks' file descriptors, except for the scheduler that must keep managing the configuration of the virtual machine [†]

The first stage ends when the fault tolerance software is sure that there is no more in-transit message and when it has calculated the maximum recoverable consistent global state of the distributed application [Conan95].

The second stage is the reconfiguration of the distributed application. The fault tolerance software transmits the list of failed pvmds and the list of failed or orphan tasks to the scheduler [‡] The scheduler creates new incarnations of failed or orphan tasks and if necessary new pvmds.

The change in the configuration of the virtual machine necessitates a change in the routing table because the new incarnations have new task identifiers and pvmds only know old ones. We solve this problem by inserting a table that does the mapping between old and new task identifiers. For sending a message a task always uses the old identifier, for routing a message a pvmd uses the new identifier and before delivering a message to a task a pvmd replaces the new task identifier with the old one. The scheduler broadcasts the map to all pvmds and at last transmits the new configuration to the fault tolerance software.

Finally, the fault tolerance software restarts the distributed application and informs the scheduler. The scheduler broadcasts a *"restart"* message to all pvmds, all pvmds start again reading tasks' file descriptors and the execution is resumed.

4 Modifications of the PVM's source code

The modifications of the PVM's source code fall into three points: the adding of new PVM message tags for the recovery protocol, the management of a table for the mapping between old and new task identifiers and the management of a lock for the stopping of inter-task communication.

We add three protocols controlled by the scheduler: one for broadcasting the new mapping table and two for setting and unsetting the inter-task communication lock. This represents six new message tags.

[†] The scheduler can still communicate because the semantic of transmission of the messages it is going to send is *"no constraint"*. See [Brzeziński95, Conan95, Conan] for explanations on the semantic of transmission of messages

[‡] The fault tolerance software had been informed beforehand that the scheduler is responsible for the configuration of the virtual machine, so of the distributed application.

The management of the mapping table necessitates a list of two-dimensional structures, two static variables and three functions: for loading the new mapping table, for returning the new identifier, for returning the old one. The first function corresponds to a new protocol and then to a new message tag. The two following ones are called before routing inter-task messages and before delivering inter-task messages to the receiving task, respectively.

The management of the inter-task communication lock necessitates one static variable and three functions: for setting the lock, for unsetting it and for testing it. The first two functions correspond to two new protocols and to two new message tags. The last one is called each time the daemon *"works"*. Recall that the master keeps reading the file descriptor of the scheduler.

Practically, 177 lines of C code are inserted in the source code of PVM and the scheduler is coded in less than 2000 lines of C code §

5 Discussion

In [León93], authors implement rollback recovery inside PVM. They make the best of synchronization routines supplied by the PVM library. For checkpointing a distributed application, they stop all tasks with a *"barrier"* so that the system is quiescent ; in particular, messages in transit must be flushed from the communication medium. The difficulty is here to determine when the virtual machine is quiescent, this because of the *"wait contexts"* ¶ When the virtual machine is stopped, pvmds give tasks permission to save their state and then proceed to checkpoint their own state. So, they take globally coordinated checkpoints. Because inter-task messages aren't logged, when a crash occurs, all tasks must rollback to the state corresponding to the last consistent checkpoint. In our solution, checkpointing is not controlled by PVM and doesn't involve pvmds, it can be coordinated or not, global (all tasks) or not. In short, the user chooses the fault tolerance of his application. In case of a machine failure, only failed and orphan tasks and failed pvmds must rollback.

The fact of keeping some tasks running while others rollback impose a change in the routing. In PVM, the routing is translated in names, i.e. the location of a task is included in his identifier. This problem is also encountered when migrating a task. In the PVM literature, different solutions were proposed for the migration problem.

§For the placement of tasks the scheduler has the same algorithm than the master.

¶See PVM's documentation to know what is a wait context.

In [Casas94, Casas95], only the migrating tasks and tasks connected to them by direct TCP connections are involved in the migration process. Direct TCP connections are closed and instead of going through these connections new messages will follow the indirect route through pvmds. Authors have designed a complex message forwarding system with home and hint maps between old and newer locations of tasks that have migrated. It is much more complex and more expensive than ours. The advantage of their solution is that a migration doesn't involve all the tasks of the virtual machine, so it scales better.

In [Stellner94, Stellner95], consistent checkpointing of PVM applications is implemented outside the pvmds. A scheduler sends a signal and a message to each task. Each task disconnects itself from the PVM system, checkpoints and then rejoins PVM getting a new task identifier. This task identifier is sent to the scheduler which sets up a mapping table. The task waits for the receiving of the new mapping table before restarting.

In [Song95], authors add a new daemon called *mpvmd* to each pvmd. mpvmd is responsible for maintaining a mapping table. The migration mechanism is implemented by a set of library routines on top of PVM, i.e. communication routines of the standard library are replaced by new routines that check by mpvmd if the identifier of the receiving task has changed. The migration call is a blocking call and don't return until all copies of the table are updated.

Because we need to modify pvmds (in order to lock inter-task communication and avoid *"notification"* messages), it seems to us easier and more efficient to insert the mapping table in pvmds. A last issue that isn't studied very often is scalability. Clearly, the solutions which frequently utilize broadcast primitives don't scale at all, this is more or less the case for all the solutions except for the one developed in [Casas95].

6 Conclusion and future work

We have enhanced PVM so that PVM applications tolerate machine failures, except from the one where the master and the scheduler are running. The solution comprises a scheduler that replaces the master for the control of the well behavior of the virtual machine and that cooperates with the fault tolerance software for the rollback recovery in case of crashes. The modifications of the PVM's source code are simple and scarce. Checkpointing is either consistent or independent and rollback recovery doesn't need to be global.

A prototype not including checkpointing and message logging has been implemented and tested. The fault tolerance software presented in the paper [Bernard94] is fully portable and then introduces a very high cost in interprocess communication. If we add this fault tolerance software under PVM, there would

be two levels of indirection between a user's task and the kernel, i.e. an inter-task message has to cross PVM and the fault tolerance software. So, we decided not to test and evaluate the fault tolerant PVM with this version but prefer implementing a new software above the Chorus micro-kernel [Rozier92]. The fault tolerance software is going to be integrated as a server into the Chorus/MiX subsystem, i.e. into the kernel. We are now preparing this new prototype.

Finally, the fault tolerance software is designed for local area networks. It can scale to large networks but large networks are then seen like a set of local area networks, each of them running an instance of the fault tolerance software. Thus, more work needs to be done for determining if there would be one scheduler per local area network or one scheduler that would manage one cluster of tasks per local area network.

References

[Bernard94] G. Bernard and D. Conan. Flexible Checkpointing and Efficient Rollback-Recovery for Distributed Computing. In *Proc. SUUG '94 Conference*, Moscou (Russie), April 1994.

[Borg89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1), February 1989.

[Brzeziński95] J. Brzeziński, J.-M. Helary, and M. Raynal. Semantics of recovery lines for backward recovery in distributed systems. Publication Interne PI-899, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes (France), Février 1995.

[Casas94] J. Casas, R. Konuru, S.W. Otto, R. Prouty, and J. Walpole. Adaptative Load Migration systems for PVM. In *Proc. of Supercomputing '94*, Washington (USA), November 1994.

[Casas95] J. Casas, D. Clark, R. Konuru, S.W. Otto, R. Prouty, and J. Walpole. MPVM: A Migration Transparent Version of PVM. Technical Report 95-002, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, Portland (USA), February 1995.

[Conan] D. Conan. Tolérance aux fautes dans les systèmes répartis : la reprise sur fautes par retour en arrière. Submitted to publication.

[Conan95] D. Conan. Un mécanisme global simple, souple et efficace et ses extensions. Rapport interne, Institut National des Télécommunications, Evry (France), Juillet 1995.

[Elnozahy92] 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), May 1992.

[Geist94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, Cambridge, Massachusetts (USA), 1994.

[Johnson90] D.B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11, September 1990.

[León93] J. León, A.L. Ficher, and P. Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, February 1993.

[Rozier92] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus Distributed Operating System. In *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, Seattle (USA), April 1992.

[Sens95] P. Sens. The Performance of Independent Checkpointing in Distributed Systems. In *3rd ACM Hawaii International Conference on System Sciences*, Hawaii (USA), 1995.

[Song95] J. Song, H.K. Choo, and K.M. Lee. Application-Level Load Migration and Its Implementation on Top of PVM. Technical report, National Supercomputing Research Center, National University of Singapore, Singapore, January 1995.

[Stellner94] G. Stellner. Consistent Checkpointing of PVM Applications. In *Proc. 1st EuroPVM User's Group Meeting*, Roma (Italy), 1994.

[Stellner95] G. Stellner and J. Pruyne. Resource Management and Checkpointing for PVM. In *Proc. 2nd EuroPVM User's Group Meeting*, Lyon (France), 1995.

[Strom85] R.E. Strom and S.A. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3), August 1985.