Fail-safe PVM: A portable package for distributed programming with transparent recovery

Juan León Allan L. Fisher Peter Steenkiste

February 1993

CMU-CS-93-124

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035.

Keywords: PVM, distributed computing, fault-tolerance, checkpoint, ABD ASAP

Abstract

Many scientific problems benefit from computations that are parallel at a coarse grain. Collections of loosely-coupled, heterogeneous computers are increasingly being applied to these problems. While individual computers are designed to be relatively reliable, a collection of several autonomous machines necessarily has a greater rate of failure. As data networks improve, and larger multicomputers are being used, rates of failure will increase.

PVM (Parallel Virtual Machine) [Sun90, GS92] is a popular software framework that facilitates message-passing network programming. We present enhancements to PVM to mask fail-stop, single-node failures from the application.

Fail-safe PVM uses checkpoint and rollback to recover from such failures. Both checkpoints and rollbacks are transparent to the application if the application does not depend on real-time events. Recovery occurs without wait for repair of the failed computer. The system does not rely on shared stable storage and does not require modifications to the operating system.

We describe the design and implementation of fail-safe PVM, present measurements of checkpoint costs, and briefly discuss shortcomings and potential avenues for improvement.

1. Motivation

Many scientific problems can be decomposed into coarse-grained modules that are amenable to parallelization on loosely-coupled processors.

Some problems (e.g, several N-body problems) require ratios of computation to communication that are large enough to admit coarse-grained parallel solutions. Other problems with less favorable ratios still benefit from coarse granularity when a sufficiently large number of independent problem instances exist.

The proliferation of fast – and often idle – desktop computers is promoting the use of networked independent machines as platforms on which to exploit such available coarse-grained parallelism. These systems vary greatly. They may be physically contiguous or distributed, dedicated or shared, administered by one or multiple organizations. More interestingly, such collections of machines may be heterogeneous, involving perhaps some supercomputers, special hardware, workstations, networks of different characteristics, and programs written in different languages.

Heterogeneity presents several problems: in what format should data be transferred? How can an application and its run-time system be managed – compiled, installed – in a diversity of environments? How can software be reused in different architectures, operating systems, languages? How can processes migrate in order to balance the load?

Heterogeneity presents opportunities as well: a diversity of computers with different strengths can better match the spectrum of computation patterns present in the different parts of an application. Further, a system that does not impose homogeneity is less particular about taking advantage of idle resources: this can be important for load balancing, as well as for systems like Condor [LLM88, LL90] and Piranha [ide92], which scavenge idle cycles from the network.

Like heterogeneity, fault tolerance is a source of problems and opportunities in networked collections of autonomous computers. Dealing with failures in sets of cooperating machines is more pressing than it is for a single computer because the rate of failure is necessarily greater. N autonomous, (i.e, independently failing) systems will exhibit a failure N times as often as a single system, on the average. This greater rate of failure annoys users, whose reliance on computers in general is based on the reliability offered by the typical computer, that is, by a single computer. But it may do more than annoy: as N and the longevity of the applications increase, the probability of the application never being interrupted by a failure approaches zero. This fact becomes more relevant as network pervasiveness and technology allow N to increase, since there is no shortage of long-lived computations.

While it is not possible to decrease the rate of failure of such distributed systems without making the individual components very expensive, it is possible for these aggregate systems to survive failures, that is, to offer availability. It is precisely the autonomous character of the components that makes possible this solution: a flexible system which draws life from many independent sources may be capable of surviving the death of any one source. This in fact means that distributed environments can not only handle their greater frequency of failures, but can be available for work more often than individual computers. This has been pointed out many times by proponents of many different distributed systems.

This report describes software infrastructure that makes such distributed systems highly available to applications. This infrastructure provides an abstracted view of the underlying hardware. Availability is achieved by transparently masking faults for applications that access resources through this model.

There are many plausible abstractions of distributed systems. Some examples of broad classes are:

general message passing (e.g, [Hoa78]) distributed objects (e.g, [DLA89]), distributed or virtual shared memory (e.g, [BZ91, BCZ91]), Linda [GCCC85, ACG86] and languages (e.g, [Tse90]). In addition, several programming styles are often used on these paradigms, e.g, bag-of-tasks, master-slave, producer-consumer, macro-pipelining, data-parallel. These styles are some times incorporated as part of the abstraction for the sake of efficiency or programmability.

Among these abstractions, message passing is the least abstract. Because of this, message passing is usually implemented efficiently[1], and can be used as a substrate on which to implement other paradigms. Further, while message passing primitives often introduce complex errors (such as data races,) they are easy to understand, well-contained, and do not require a cultural leap on the part of the programmer. Perhaps for these reasons, message passing libraries are very popular. Among the most popular is PVM (Parallel Virtual Machine) [Sun90, GS92]. PVM, together with UNIX, and the TCP/IP and UDP protocols, provides an abstraction consisting of a set of processes that exchange messages, synchronize, and create and destroy each other.

The prototype described herein is implemented for the PVM model. A message-passing model is chosen as a first step in supporting other models. PVM is chosen among these because of its public availability, popularity and portability.

1.1. Goals

Through fail-safe PVM we seek to explore fault tolerance for distributed computation from a practical perspective. The fail-safety features are considered to be add-ons to an existing model for distributed computing, viz PVM in the prototype. We desire the following from the add-ons, with respect to the base model.

- *Application Independence.* Fail-safety is provided for arbitrary model programs.
- *Application transparency.* Failures are invisible to applications that do not bypass the model.
- *Compatibility.* The interfaces presented by the model and the modified model are compatible.
- *External to operating system.* The implementation requires only the standard OS interface. This makes it portable, which in turn facilitates heterogeneity and the constitution of systems containing many machines from different administrative domains.
- *Minimal overhead.* Minimize overhead during regular execution.
- *Practicality.* The whole system is considered. In particular, the existence of distributed stable storage is not assumed.

And we are willing to accept the following:

- Limits on the types of failures used. We expand on this in the following section.
- Limit on the number of simultaneous failures.
- Loss of efficiency, resulting from unwillingness to modify the operating system.

[1] even on shared memory hardware

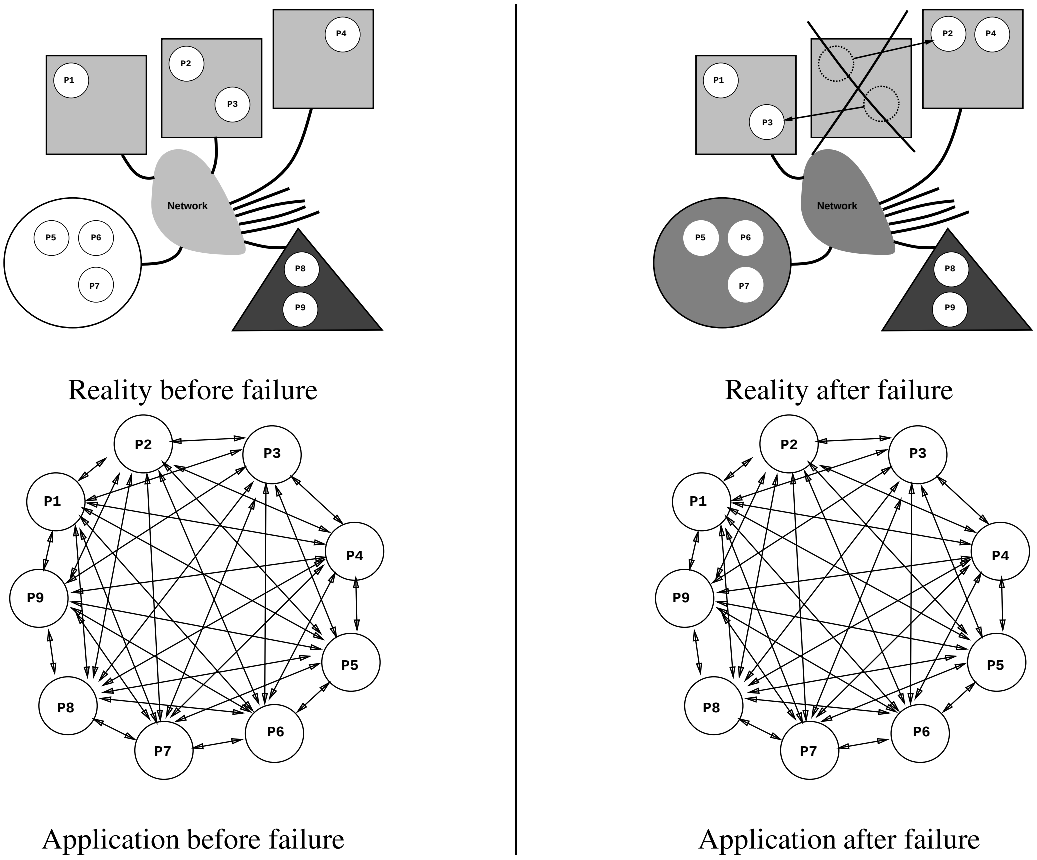

Figure 1: Failure masking

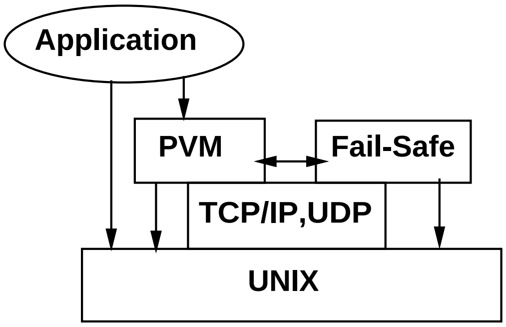

Figure 2: Architecture of PVM and fail-safe enhancements

2. Fail-safe PVM

Fail-safe PVM is an enhanced version of PVM. The enhancements serve to mask certain failures from the application, without changing its view of the system.

We make no attempt to deal with arbitrary (byzantine) failures. We require that failures be of the fail-stop type. These failures cause complete failure of the node in which they occur. Throughout this report it is assumed that the operating systems installed in each node guarantee this condition. In particular, nodes do not send messages with incorrect contents. Nodes are in general assumed to be cooperative, i.e, secure and not malicious.

We also require that failures not be deterministic. Deterministic failures can only be eliminated by changing the application, i.e, by eliminating some of the work that triggered the fault. In general, fail-safe PVM does not detect or correct software faults.

The current implementation of fail-safe PVM can mask at most one simultaneous failure. This limitation is arbitrary. A few modifications would suffice to allow an arbitrary number of simultaneous failures. While designing this and future versions we are keeping in mind the following goals: that there be no structural impediments to masking multiple failures, and that the cost of tolerating n failures be linear on n.

Fail-safe PVM masks failures through checkpointing and rollback. Checkpoint and rollback provide retroactive process migration: they can be combined to make it appear that all processes are migrated from a machine when it is about to fail. Rollback provides this foresight. Checkpoints enable the rollback.

A checkpoint is a copy of the instantaneous state of a system. A single process[2] can checkpoint itself at any point: when its single thread of control is checkpointing, the state of the process does not change, and the copy of memory (and supporting kernel structures) recorded will reflect an instantaneous snapshot.

A set of communicating processes, taken as a single entity, is much harder to checkpoint. Each process has a different perspective on the state of the whole system, at any given time. More importantly, it is impossible for a process to know when other processes are checkpointing, without introducing a delay during which the global state may change. Allowing separate processes to checkpoint at their convenience can result in rollbacks to inconsistent states, where some messages may appear to have been sent but never received, or even appear to have been received without having been sent. All these problems have been amply discussed in the literature, for instance in [Lam78, CL85].

Fail-safe PVM checkpoints a PVM session by forcing a global synchronization before allowing a checkpoint to proceed. After such a barrier, all processes are guaranteed to be stopped and the global state is guaranteed to be consistent and unchanging. The set of local checkpoints taken under these circumstances is a valid snapshot of the state of the session.

Our objectives are pragmatic. Many more complicated checkpointing procedures (some with message logging) have been proposed. This globally coordinated checkpointing approach has the practical advantage of being simple. In section 6 we position this simple checkpointing procedure among some of the approaches suggested in the literature, and in section 7 we speculate on the benefits and drawbacks of this choice. In light of the different assumptions and requirements, it is premature to clearly state which approach is best suited to our goals.

[2] By process we mean a single thread of control

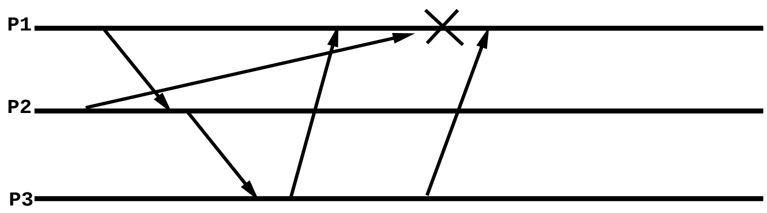

Figure 3: 3 processes interacting

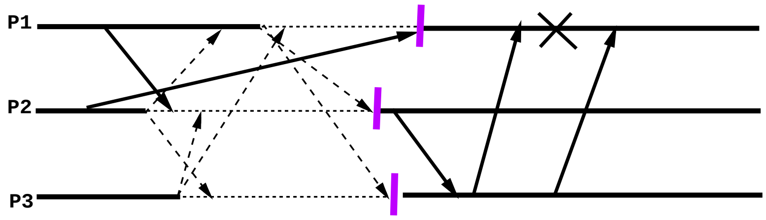

Figure 4: Globally coordinated checkpoint

3. Design

We first present a general design of a globally synchronized checkpoint and its associated recovery (rollback) procedure. We then refine the design in order to apply it specifically to PVM.

3.1. Globally coordinated checkpoint/recovery

3.2. Checkpoint

Figure 3 illustrates a hypothetical interaction among three processes. The solid horizontal lines denote process execution. The horizontal axis represents real time. The cross represents a failure on the process on which it is marked.

Figure 4 illustrates how a checkpoint is taken. Horizontal dashed lines denote processes idled by synchronization. Dashed arrows represent communication required for synchronization. Short vertical bars denote the times at which local checkpoints are taken.

Figure 4 illustrates three points. The first is that the synchronization may start in different processors at very different times. The dashed arrows might extend horizontally for as long as many tens of thousands of cycles. This is the cost of synchronizing. Second, the barrier is not necessarily completed as soon as all processes enter the synchronization phase and report the fact to each other. Before the local checkpoints can be taken, the system must quiesce; in particular, messages in transit must be flushed from the communications medium. This is required because messages can only be checkpointed at the boundaries of the medium, where the stable storage is. `P1` in figure 4 experiences this flushing delay while waiting for the message from `P2` to arrive. In systems that, unlike the one in the figure, guarantee channels to be reliable and order-preserving, synchronization implies quiescence if the synchronizing traffic proceeds over the same channels as regular messages. Finally, the processes must be able to receive and process messages while quiescing.

The state of any process exists partially in the operating system. The operating system structures

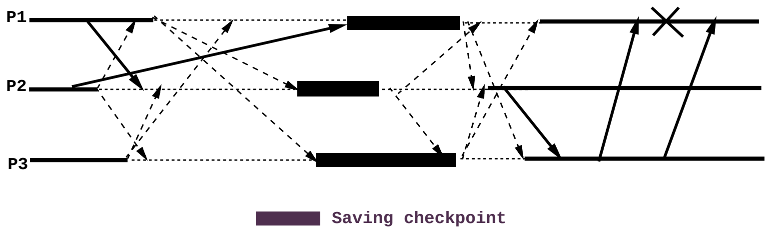

Figure 5: Non-instantaneous local checkpoints

supporting the process have to be reflected in a checkpoint of the process. This problem of checkpointing operating system entities becomes intractable if the process is executing a system call. The operating system stack (call record + local variables) is likely to contain much transient information about the state of the external world: disk heads, network drivers, etc. While this information can be checkpointed, the external configuration it encodes is hard to replicate upon rollback. This is the problem of reconstructing the kernel stack.

To avoid this problem, we require that no process be checkpointed while it is executing a system call. In practice this means that checkpoints cannot be made while a process is synchronizing on behalf of the application, or when it is blocked waiting for reception of a message. These application operations (i.e, synchronization and message reception) must be interrupted when the system reaches a quiescent state. This interruption must be recorded in the checkpoint so that the operation can be restarted when the checkpoint is reinstated as part of recovery.

The last remark concerning the checkpointing procedure illustrated in figure 4 is that local checkpoints, shown by vertical bars in the figure, do not occur instantaneously. In fact, the time required may be quite long, as it spans a copy of the entire process state to a remote site. Typically, it also spans a copy to disk, whether local, remote, or both.

To ensure that failures are properly masked during this period, it is necessary to have access to two checkpoints per process. This is a sufficient and necessary condition [KT87]. The older of the two checkpoints stays current until the newer one is completely written. Before the new checkpoint is completed, failures are masked by rolling back to the older set of checkpoints. It is essential that the notion of current checkpoint be updated synchronously in all processes. Thus, a second global synchronization is necessary. Figure 5 reflects this final barrier.

It is still possible for a failure to occur after the final synchronization takes place, but before all processes have updated their notion of current checkpoint. Live processes will eventually carry out the update. Failed processes will carry out the update when restarted from the recently completed checkpoint. Failures between the barrier and the update can be handled by ensuring that the recovery procedure cannot be initiated in that interval. This assurance is easily provided because it can be enforced locally.

3.3. Recovery

The checkpointing procedure described above records a global snapshot of the state of all processes by recording local snapshots in a manner that guarantees their consistency. The system can be rolled back to one such snapshot by rolling back each process to the last completed local snapshot.

In doing so, two issues are important. First, any messages in transit at the time of recovery must be flushed. Second, no rolled back process should restart before all processes have rolled back. Failure to meet the first condition may result in messages going backwards in time [3]. Failure to meet the second condition may result in the loss of post-rollback messages, which will instead be received by abandoned computations.

While the second condition requires a synchronization, the first one only requires that we ignore messages sent before the rollback as they arrive. If the communication channels are reliable and order-preserving, this can be done by sending a marker through each channel during recovery. In general, the same effect can be achieved by destroying all existing channels and establishing new, empty ones. This may be somewhat expensive, but a great deal of performance can be sacrificed during the relatively rare instances of recovery.

3.4. Failure detection

A question remains of when to trigger the recovery procedure. It is impossible to distinguish a distant process from a failed process: both may appear to be unresponsive. In practice, the distance to a process is determined and allowed to change (slowly) through dynamic time-outs and retries. While it is still impossible to distinguish a process whose latency increases suddenly from one that fails, the parameters can be chosen judiciously so that this occurs infrequently. When it happens, it should always be correct, if not optimally efficient, to conservatively declare a failure. Links to processes whose failure has been declared should be severed.

3.5. Globally coordinated checkpoint/recovery in PVM

In PVM, processes communicate and synchronize through daemons, one of which is present in each computer included in the distributed system. Daemons coordinate global operations and processes carry out local computation. This multiplicity of agents complicates the clean addition of features, but PVM's architecture simplifies our work in at least three ways.

Failures occur without prior notification, and are thus asynchronous. A system that strives to mask failures transparently cannot expect an application (or its library, running under the same thread of control,) to poll or listen asynchronously for failures. A failure daemon is required. This logical daemon can be incorporated into the existing PVM daemon, and use the existing inter-daemon communications infrastructure.

PVM's architecture also facilitates a solution to the kernel stack reconstruction problem. When PVM processes are blocked waiting on reception of a message or conclusion of a synchronization step, they are blocked waiting for a response from the daemon, not from the operating system. The daemon is then a convenient point at which to cancel pending operations before a local checkpoint is taken.

Finally, PVM implements reliable and order-preserving communication channels. This greatly simplifies the design of flushing and synchronization algorithms for checkpointing/rollback.

---

[3] In fact, in non-deterministic applications these messages may never be sent after the rollback.

3.6. Checkpoint

Checkpointing involves the following steps:

1. Some daemon commands the other daemons to flush inter-daemon communications. This requires a synchronization during which all daemons process incoming messages and cancel or complete requests from application processes.
2. Daemons interrupt processes as their requests complete (or are cancelled,) commanding them to:
   (a) flush their virtual circuits, and
   (b) wait for permission to checkpoint local state.
3. Daemons synchronize amongst themselves.
4. Daemons give user processes permission to save state, along with a temporary file name, and then proceed to
   (a) checkpoint their own state, and
   (b) wait for all processes to finish checkpointing.
5. After their virtual circuits are flushed, processes checkpoint their state.
6. Daemons move all local checkpoints to a remote site, if they have not yet been saved on a global file system.
7. Daemons synchronize.
8. The recently completed temporary checkpoint becomes permanent in all daemons.

3.7. Recovery

There is a question of how much state should be reused during recovery. One extreme is to kill all daemons and processes (except one recovery director) and to restart them with the checkpoints. This has the advantage of simplicity, but takes a long time (which is linear on the number of nodes and not on the number of failures.) An alternative is to synchronize all processes so they stop sending messages, restart the failed processes only, rollback all processes, and then synchronize again before allowing them to continue. This is much more complicated, and has the disadvantage of having to remember what processes terminated and started since the last committed checkpoint.

The compromise chosen for this prototype is to terminate and restart all processes, but restart the failed daemons only. This compromise may turn out to be faster than the more complicated scheme as it eliminates the need for several synchronization operations.

The sequence of events during recovery is:

1. Some daemon noticing failure becomes the recovery master. It then informs all other daemons of the failure. Daemons are totally ordered, and conflicting claims for master status are resolved in favor of daemons further down in this order.

2. Daemons terminate all local processes.
3. Daemons roll back themselves and local processes, and then synchronize.
4. Recovery master reads the checkpoint of the failed daemon.
5. Recovery master commands other daemons (of its choice) to restart the processes of the failed daemon. The restoration of failed processes entails the redistribution of their checkpoints, so sufficient backup copies are still available for future failures.
6. Recovery master sends processes, through daemons, permission to continue.

A system that is resilient of n failures maintains the availability of n copies of all checkpoints. Immediately after a failure, and until the checkpoints are redistributed in the recovery that follows, only $n - 1$ copies of the checkpoints of some processes will be available. In that interval the system is vulnerable to $n - 1$ failures. This window of vulnerability can be reduced but not eliminated. We note that this window is not a result of the reliance on a master. Failures during the recovery procedure will trigger other recovery procedures during which other masters will be chosen. Some of these double faults will be recoverable.

3.8. Failure detection

We conservatively define a machine's failure as a daemon's failure. The daemons declare a failure when they can agree that they have lost contact with any daemon. In our prototype, daemons do not so much agree as are forced to accept the criterion of a self-elected recovery master.

4. Implementation

In this section we briefly touch upon some issues regarding the implementation of the mechanisms used in the procedures described in the previous section.

4.1. Communications

We distinguish between synchronization and communication requested and performed on behalf of the application and those performed on behalf of the checkpoint/rollback modules. The initial synchronization step of checkpointing requires that the system be flushed of pending work (requests in transit) before the barrier completes. This step requires knowledge of the state of the application communication channels. The implementation uses the channels maintained by the daemons on behalf of the application to carry the daemon synchronization traffic. These channels are implemented by PVM to be reliable and order-preserving. This guarantees that all application traffic directed to a daemon has arrived before that daemon can be informed of the completion of the barrier.

The interaction between the application processes and the daemons for the purposes of masking failures is asynchronous. Checkpoints and rollbacks are initiated asynchronously, perhaps at a point where a channel between a process and its daemon is occupied with a pending request. Traffic for checkpoint and recovery between processes and daemons must then use channels different from those used by vanilla PVM.

The duplication of channels exacerbates one of the limitations of PVM, namely the profusion of opened communication resources (a.k.a file descriptors in UNIX.) We have limited the damage done in this respect. All communication between application processes and daemons is multiplexed on the same sockets through use of UNIX signals, queues, and a state machine grown organically (i.e, developed in an ad-hoc way.) Also, the notification of failure is carried over TCP links that PVM maintains for daemon coordination.

The notification of failure is special for another reason. Essentially, it is a broadcast. In networks with more than one physical channel (i.e, not ethernet or token ring,) this broadcast is typically carried out on a spanning tree (e.g, recursive doubling as used by PVM.) Such a spanning tree, however, may have been divided by the failure that is being broadcast. The information in the message must be used to change the way the message is routed.

4.2. Master daemon

Current PVM implementations distribute information about the system (e.g. process locations, architecture types) over the daemons. Some information is replicated, some is privately owned and cached remotely for read-only access. Unfortunately, critical sections for updates are all arbitrated through the daemon started by the user.

This master daemon, then, is a vulnerable point of failure. Its failure can still be masked through a more complicated procedure in which the other daemons negotiate to choose a new master. While this is practical, it may not be advisable for all programs. A visualization program, for example, should be stopped when the display becomes unavailable. A parameter, set for each session, should switch fail-safe PVM between these two modes. The current version of fail-safe PVM does not have this feature.

More broadly, the reliance of PVM on a master (if only for process creation and destruction) poses alternatives in the implementation of global synchronizations and broadcasts. In a system without an arbitrator, a combining tree can be used to implement these operations with $\mathcal{O}(N)$ messages in $\mathcal{O}(\log N)$ time. An arbitrator allows them to be implemented with $\mathcal{O}(N)$ messages in $\mathcal{O}(1)$ time. We have preferred to not assume an arbitrator, in the hope that this work will be partly applicable to systems other than PVM.

4.3. Checkpoints

Daemon checkpoints have to be understood by the recovery master, which must recover the failed daemon's processes table and backup sites. We use a known format for these checkpoints so that this can be achieved. This format is independent from the architecture and includes only information necessary for rollback.

The checkpoints of the application processes must be transparent, and are therefore architecture-dependent. A checkpoint of a UNIX process requires saving of the execution context, the text of the program (code and constants), the stack and the heap. We use the standard `setjmp()` and `longjmp` library routines to portably save the execution context. We save the stack and the heap into a file simply by writing the appropriate memory addresses with a `write()` system call. We do not save the program text at all, but instead read it upon restart from the executable file, whose name we save with the stack, heap, and execution context, in the checkpoint file.

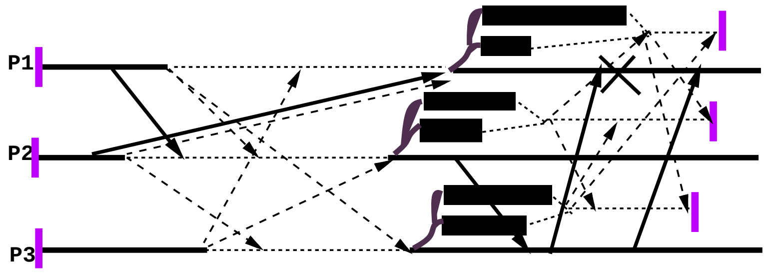

Figure 6: Copy-on-write checkpoint

4.4. Asynchronous checkpoints

The longest phase of the checkpointing procedure is the writing of checkpoints to replicated stable storage. Each checkpoint is written to a local disk, and then written to $m-1$ remote disks, with m being the number of simultaneous failures that the system must mask. Disks are slow, some networks are slow. In single channel networks (such as ethernet and token ring) the performance of this phase is further hurt by network congestion.

Writes to local and remote disks could be eliminated: the checkpoint could be stored in the memory space of the daemons, for instance. However, copying of the checkpoints over the network cannot be avoided if they must remain available after failures. Since writing to disk mostly overlaps the network transfer, these disk writes are less expensive than it might seem. Given this, we have not implemented this optimization.

But the creation of stable checkpoints remains a bottleneck. To palliate its effect on the overall performance of applications, we have implemented copy-on-write checkpointing. After the system is quiescent following the first checkpointing synchronization, the application is allowed to continue execution. The checkpoints are taken concurrently. As the application continues, it modifies its state. When the application touches a page that has not yet been checkpointed that page is copied before the modification is allowed to take place, making the old version available to the checkpoint. Figure 6 illustrates copy-on-write checkpointing. After the initial flushing of messages in transit the application, represented by a solid horizontal line, continues executing. The state of the application is also copied into the local disk and into a remote disk. The two copies and the application proceed concurrently. The new checkpoints are committed without interrupting the application.

This modified, copy-on-write checkpoint still consumes disk and network resources. This is likely to slow down the application, but not stop it completely as in the more naive approach. In section 5 we present preliminary results that suggest that this more complicated approach is beneficial.

Copy-on-write checkpointing is implemented with copy-on-write `fork()` system calls, which are available in most versions of today's UNIX systems. This preserves the user-mode character of fail-safe PVM.

5. Performance

It is premature to categorically announce the overhead imposed by the fail-safety enhancements on PVM, for two related reasons. The first is that we have not yet been able to examine fail-safe PVM under a

checkpoints	Synchronous checkpoints	Copy-on-write
`Comm`	7.0 s.	1.1 s.
`NoComm`	5.8 s.	1.2 s.
`Mandelbrot`	0.9 s.	0.7 s.

Table 1: Overhead imposed by checkpoints

sufficient number of production runs. The second is that we haven't executed fail-safe PVM long enough to extract statistically reliable figures.

The performance of the checkpointing procedure is highly dependent on the network load. The numbers we present in this section were obtained on computers physically distributed through a local area network composed of tens of subnets, and linking nearly two thousand hosts. The load on this network was highly unpredictable. The deviation in our measurements was commensurate. However, we believe a qualitative indication of costs can be extracted from them.

We first measured the overall overhead of checkpointing, and then tried to identify the phases of the procedure that contributed the largest portion of the cost.

We used three synthetic benchmarks in evaluating fail-safe PVM. Each of the N processes of the `Comm` benchmark sends $\frac{5000}{N}$ messages to the other $N-1$ processes, each message containing 500 floating point numbers. `Comm` does not compute anything, beyond the overhead incurred by PVM to communicate. It is intended to measure the effect of communication on the overhead of checkpointing. The size of its state (i.e, its checkpoints) is small, around 100K per process.

`NoComm` does not communicate at all, but has a larger state size of about 1.2MB per process. We would have liked to explore the effect of large checkpoints with larger state sizes, but the amount of temporary space available in the disks of some machines limited us. Minimizing required disk space is an issue that we have not yet addressed.

Finally, `Mandelbrot` is the demo program included in the PVM distribution. It computes on a master/slave paradigm, distributing tasks to slaves as they conclude the previous ones. Task messages are small, result messages carry 500 floating point numbers. The ratio of computation to communication is approximately 2:1 when using PVM among 4 SUN 4's and 4 DECStation 5000's connected through a large local area network.

We run these programs on 8 machines under three configurations of PVM: without checkpointing, checkpointing synchronously every 30 seconds, and with copy-on-write checkpoints every 30 seconds. The measurements were repeated at different times of the day, multiple times.

To determine the total overhead of checkpointing, we subtract the average execution time of the runs with no checkpointing from the average execution time of the runs with checkpointing. We then divide the result by the average number of checkpoints taken. This yields an estimate for the time it takes to carry out a single global checkpoint. This is better than a number relative to the overall execution time because the interval between checkpoints is specified separately for each session, and will usually be greater than 30 seconds. Table 1 shows the results.

To estimate where the overhead is spent, we divided the checkpoint into the phases illustrated in figure

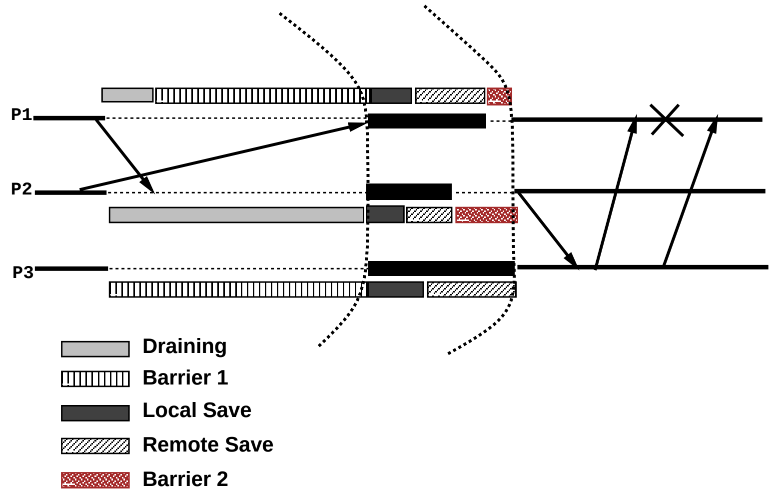

Figure 7: Phases of checkpoint

7. Draining occurs between the time a checkpoint is started and the time at which a daemon has no work pending for the local processes. The first barrier is the interval following draining during which a daemon waits to be notified of the quiescence of the system. During local and remote save phases the daemon waits for the writing of checkpoints to be completed into local and remote disks, respectively. The second barrier is the interval during which a daemon waits to be notified of the completion of all saves.

Each daemon collects local timestamps to measure how long each of these phases is. It is difficult to present these numbers together in a meaningful way. Daemons may enter and exit phases in no particular order; local clocks are not synchronized. The only guarantee we have regarding ordering in real time is that the first daemon to exit a barrier does so not before the last daemon enters it.

We seek to measure the contribution of synchronization and checkpoint saving to the overall cost of global checkpoints. The cost of saving checkpoints can be further divided into that attributable to the slowness of disks and networks, and that caused by the contention for disks and networks produced by the saving phases.

For each checkpoint, we subtract the saving time of the host for which it is longest from the total checkpointing time of the host for which this is longest. We then average the numbers so obtained for each run. We use this result as an estimate of the cost of communication during checkpointing.

To understand why, consider the case where communication is instantaneous. In this scenario, all hosts start the checkpointing sequence and complete barriers at the same real time. Also, at least one host completes the barrier as soon as it starts it: the length of the corresponding barrier phase of figure 7 for that host will be 0. This host will also exhibit the longest saving phase, since this phase is started by all hosts at the same real time. The total time to take the checkpoint, then, will be the same in all nodes, and will be equal to the length of the longest saving phase.

In the more realistic case of non-instantaneous communication, the cost of communication can be defined as the result of subtracting the cost of checkpointing with latency-free communication from the actual cost of checkpointing. This is what we show in table 2 as a percentage of the average total checkpointing cost.

Note that this estimates the cost of all communication, including the cost of broadcasting the initiation of the checkpoint sequence. The cost of synchronization alone cannot be estimated without synchronizing

	Comm	NoComm	Mandelbrot
Synchronous	67 %	6 %	24 %
Copy-on-write	77 %	3 %	45 %

Table 2: Cost of communication

	Comm		NoComm		Mandelbrot	
	Local	Remote	Local	Remote	Local	Remote
Synchronous	516	800	1596	6649	561	1064
Copy-on-write	424	965	1828	8685	476	1063

Table 3: Cost of checkpoint writing in ms.

the node clocks. This we do not do because the cost of communication is expected to be dominated by synchronization, and not by a single broadcast.

To estimate the cost of saving checkpoints we calculate the average mean length of the local and remote save phases of figure 7. To estimate the cost of contention for disks and network generated by the saving phases, we measure the total checkpoint saving interval for the `NoComm` program running on systems with 2, 4, and 8 hosts. We choose `NoComm` because it has the largest state.

The results are shown in tables 2, 3 and 4.

Tables 2 and 3 show that most of the cost of checkpointing `Comm` is due to synchronizing, and little of it is due to saving checkpoints. This is expected of a program with small state size and busy communication channels. The converse is true of `NoComm`.

We expect the synchronization costs of applications to lay somewhere between the synchronization costs for these synthetic benchmarks. We expect the saving times of real applications to be more in line with the saving costs of `NoComm`, or to be larger. This is in accord with the estimate that applications will be less extreme than `Comm` an d`NoComm` regarding communication requirements, but will in general have fairly large working sets.

Synchronization is currently naively implemented, and has a complexity of $\mathcal{O}(N \log N)$. This harms prospects for scalability, as shown by the first line of table 4. We have plans to implement synchronization with a combining tree, in $\mathcal{O}(\log N)$.

The second line of the table also points out another problem regarding scalability. In that line, the increase in times required to save checkpoints is seen to increase with the number of nodes. When backup

	2	4	8
Barrier 1	20	64	178
Save	7076	8816	16000

Table 4: Scalability (lack thereof)

copies of checkpoints are judiciously distributed each disk stores a constant number of checkpoints. If the network is not overloaded, there is then no reason for this increase in saving time. During the measurements above, the checkpoints were distributed uniformly among the disks. From a separate experiment [4] we know that the network has spare capacity. We have yet to understand the source of this unexpected result.

Clearly, we have not performed enough experiments to safely assert anything about the scalability of fail-safe PVM. Enthusiasm for better results in this regard is weak because scalability imposes limits on PVM similar to those it imposes on fail-safe PVM. However, scalability of the different methods of masking failures is important to us in general, and may prove to be the nemesis of the current globally coordinated method.

6. Comparison with other work

One of the main promises and pitfalls of distributed systems has been the potential to tolerate and suffer faults. Much effort has been spent trying to ensure that distributed computations are "correct" in the presence of failures.

"Correctness" often refers to the consistency of results. Transaction systems, for instance, are concerned with preserving a consistent state of the data base to which they control access. Occasionally, this prevents progress: two-phase commit protocols must wait until failed disks come back in line. In our primary domain, scientific computation, data consistency is guaranteed by the application. Our main concern is then to minimize the performance degradation in the presence of failure. We do not accept solutions that require waiting for a computer (or its disk) to be reconnected to the network.

We also require that our solutions be transparent, or invisible, to the application. A program that executes in PVM should execute in fail-safe PVM without changes. Transaction systems, mechanisms such as recovery blocks [Ran75] and those used in Tandem's non-stop kernel [Bar81] are some examples that lack transparency [5].

The methods discussed in this section can be ordered on a linear spectrum in which overhead during regular operation is traded-off against cost of recovery. We start with the method of process replication, on the end of efficient recovery and high overhead.

Transparent systems that are tolerant of faults rely on replication of resources. In some cases the replication is complete: m processes execute concurrently for each process required by the application. If one fails, the others can continue. This approach imposes an overhead of at least $(m - 1) \times 100$ per cent. To this overhead, one must add the overhead of additional communication: each point-to-point communication becomes m m-way multicasts. Furthermore, to assure that all replicas are in the same state, they have to be executed in virtual lockstep. This typically requires that the multicasts (which are semantically point-to-point messages) be causally ordered [**?**]. Much can be done to optimize this traffic, given its regularity. However, the cost imposed on the application in the absence of failures is still high. In favor of this scheme, we point out that the cost of recovery from failure is very small. This makes it suitable for real-time environments, where poor performance is preferable to unpredictable performance.

---

[4]The same machines were programmed to contact about 1000 other computers through a low level protocol (ping) in an attempt to flood the network. The network was monitored by special devices. The load on the network never surpassed 10 %, even though the machines were losing incoming packets. We speculate that the protocol stack, and not the network, is the bottleneck.

[5]necessarily, since most handle many more types of failures, e.g, software faults.

More unpredictable but probably better (on the average) performance is available from systems that replicate only part of the resources, and that then use more complicated recovery procedures to reinstate the rest following a failure. In particular, the "virtual" replicas do not consume cycles on behalf of the application: the state of the application is partly replicated and partly rebuilt. This is done with a mixture of checkpointing/rollback and message logging/replay.

A complete taxonomy of such methods is beyond the scope of this section. We briefly sketch the following checkpoint/logging methods: pessimistic logging, uncoordinated checkpointing, (globally) coordinated checkpointing and optimistic logging.

In pessimistic logging (also known as synchronous journaling), the contents of all messages are transmitted to the virtual replicas of a process before they are delivered to the process. In practice, this means that messages are copied to a local and to a remote disk before control is returned to the process. Periodically, the process is checkpointed so that a failure triggers message replay for no longer than the checkpointing interval.

Synchronous logging of messages is analogous to the m-way multicast necessary in the process replication scheme. If m copies of the message log are kept (for availability,) the messages have to be distributed to these copies in a causally ordered manner. Whether a running process or a disk is receiving these messages is irrelevant as far as this requirement is concerned. While this ordering is relatively easy to guarantee in broadcast networks or buses (e.g, ethernet,) it requires considerable overhead in general.

Another shortcoming of synchronous message logging, and, indeed, of all logging mechanisms, is that care must be taken to ensure that the computations are deterministic between messages. This is not a problem not only for applications that use randomizing algorithms, but with almost any application that interacts with the operating system: unless the operating system participates closely in the recovery, any interaction with the OS will make a computation non-deterministic. For instance, if the application acquires a process identification number (pid in UNIX parlance) from the Operating System, the OS must log the actual pid returned, and reuse it in the same manner should a rollback occur. This and other similar problems are discussed in [BBG$^+$89].

On the positive side, synchronous logging allows a process to fail and recover without disturbing the rest of the system.

The elimination of logging results in the method we call *uncoordinated checkpointing*. Each process takes periodic checkpoints of its own accord. Upon failure, a set of checkpoints, one per process, is chosen, and all processes – not just the failed ones – are rolled back. A process cannot roll back in isolation, because in general such a process will have had interacted with other processes since its last checkpoint: an isolated rollback would produce an inconsistent global state in which messages might have been sent but would never be received, or received without having been sent. Neither will it suffice to roll back all processes to the last checkpoints. Any arbitrarily chosen set of checkpoints may contain an inconsistent global state. The processes must be rolled back to a well chosen set of checkpoints. Setting aside the problem of making this choice, this method is susceptible to the domino effect.

Figure 8 illustrates the domino effect. Since messages in transit cannot be checkpointed, the only set of checkpoints that would make the system roll back to a consistent global state is the one shown by the vertical recovery line. The application would have to restart from the beginning. This situation is not as unlikely as it may seem in the figure. Typically, the interval between checkpoints is a lot greater than the interval between messages, and so the likelihood that a potential recovery line is not crossed by any messages is small.

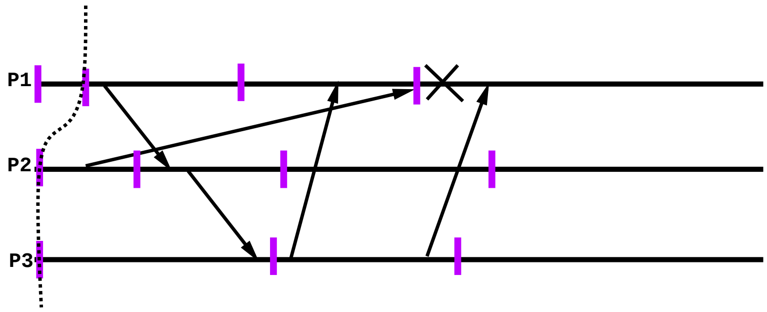

Figure 8: Scenario with domino effect

Coordinated checkpointing avoids the domino effect by synchronizing the processes, ensuring that a set of recent local checkpoints contains a consistent state. The synchronization can be done globally, in which case the resulting global checkpoint can be used to recover from any failure. Alternatively, each process can be allowed to decide when to create a checkpoint from which to recover its failures. In the latter case, only those processes that have interacted (transitively) with the checkpointing process since the previous checkpoint need synchronize [LB88].

When the synchronization is not global, some overhead is required to maintain information about the interaction among processes. The synchronization may also proceed more slowly than in the global case, since the dependency information is inherently distributed. This synchronization may be repeated as each process initiates a checkpoint in order to cover its failure. Results from other process' checkpoints can be cached, greatly decreasing the actual number of local checkpoints taken, but some unnecessary rounds of message exchanges will take place. Also, the cost of synchronizing globally may not be considerably higher than the cost of the causal synchronization necessary for non-global coordination. Suppose that processes are divided in cliques, among which communication is rare. In this case, non-global coordination synchronizes only one clique most of the time. However, if there is little communication among the cliques, it will not be much more expensive to synchronize among the cliques at that time. If there are no cliques, the non-global case reduces to the global one.

Globally coordinated checkpointing, however, is deficient in terms of scalability. The limits in this regard come from two sources. First, the synchronization steps cannot be implemented in better than $\mathcal{O}(\log N)$ complexity, or with less than N messages. For networks such as ethernet and token ring a synchronization requires $\mathcal{O}(N)$ time, if the broadcast at the physical layer is used. More naive implementations would require $\mathcal{O}(N^2)$ time. In all networks, knowledge of the network topology is required to do better than the naive implementation. Second, after synchronization takes place, local checkpoints start concurrently in each node. mN checkpoints will be concurrently written onto disk, where m is the number of simultaneous failures tolerated and N is the number of nodes in the system. $(m-1)N$ of these will have to traverse the network. If the network is not saturated, the bottleneck is at the disks: $m\frac{N}{d}$ checkpoints will be concurrently written to each disk, with d the number of disks. If the network is saturated, the $(m-1)N$ checkpoint transfers will be serialized in some way. Concurrency and performance are wasted as N increases, d decreases, and the network bisection bandwidth decreases. These shortcomings are bypassed by not coordinating checkpoints while carefully avoiding the pitfalls of uncoordinated checkpointing and pessimistic logging. This is the goal of optimistic logging.

Optimistic logging logs messages onto disk asynchronously. The messages are sent or received while, or even before, the log entry is completed and written. This delay implies that processes may not be completely recoverable, since some messages sent or received before a failure may not have been logged prior to it.

Any lost events must be regenerated, even those generated by unfailed processes in the original execution. This is achieved by rolling back such processes to a point before such events occurred.

The complexity of optimistic logging lies in determining what these points are. It requires that complete information on the causal dependencies among the processes be kept and updated during regular operation. It requires a round of negotiation during recovery in which this distributed information on inter-processor dependence is exchanged until a consensus is reached.

This consensus is also needed to determine the points to which it will not be necessary to roll back. As checkpoints and log entries are committed in all processors, previous points of the execution become committed too, in the sense that no failure will require a roll back to these points. Determining such committed computations is crucial for recycling the storage used by checkpoints and logs. The rounds of information exchange required for this seem to be roughly similar to a global synchronization, and must be carried out during regular operation.

In contrast, globally coordinated checkpointing stores at most two checkpoints per process, does not store log entries, imposes lower overhead during faultless execution, and is simpler. On the other hand, it is not a scalable method, its recovery causes longer rollbacks, and recycles storage upon every checkpoint, which is more often than necessary when disk space is not limited.

These two approaches seem vastly different, and yet may be more closely linked than anticipated. How different is optimistic logging from globally coordinated checkpointing with the implementation of incremental and asynchronous checkpoints?

We have already mentioned that the first implementation of fail-safe PVM uses globally coordinated checkpoints. Optimistic logging is a newer, more complicated method that could improve the performance of the fail-safety enhancements. Quantitative analysis is required to determine the extent to which this is true.

7. Future work

We would like to obtain better estimates of the overhead imposed by checkpoint/recovery. Because fail-safe PVM becomes more useful as the size of the virtual machine increases (and failures become more frequent) it is important that we ascertain the extent of its problems with scalability. We will measure the performance of checkpointing systems with substantially more machines. In an attempt to obtain more credible numbers, we would also like to measure the impact of fail-safe PVM on real applications. We hope that this report will trigger some cooperation in this respect.

It is also important to measure the performance of the recovery procedure. Heretofore we have assumed that recovery is so infrequent that its cost is of no great concern. However, an expensive recovery procedure and/or a rollback into the far past can block progress of an application. This may happen if the mean time between failures is smaller than the computation time lost to the recovery and rollback.

In designing fail-safe PVM we have chosen a level of abstraction in which to implement checkpoint and recovery. We have chosen to remain transparent regarding services provided – transparent to the application – as well as regarding services used – impervious to the implementation of operating systems, network protocol stacks, and lower layers of PVM. While this choice is meritorious from the software engineering perspective, it may prevent improvements in performance.

It could be worth our while to investigate modifying some of the services used. For example, we could modify PVM's implementation of reliable order-preserving channels. Control over these mechanisms would simplify the synchronization steps. Messages waiting in queues would not need to be flushed; the already implemented mechanisms for retransmission of lost messages and detection of duplicate ones would even obviate the need to flush messages in flight through the network. Or, a second example, we could augment PVM message headers to carry a checkpoint generation number which would count the number of checkpoints taken by the sender. This would enable the flushing phase of the checkpointing procedure to proceed asynchronously with the execution of the application [EJZ92].

If we allowed our modifications to encompass the operating system we could also efficiently implement incremental checkpoints by using the information contained in the processes' page tables. This does not greatly harm prospects for portability, as newer versions of UNIX, notably the ones from OSF, allow the management of memory to be carried out outside the kernel, on a per-process basis.

It would also be instructive to explore the relaxation of application transparency. Most applications synchronize occasionally. Checkpoints taken as the application synchronizes would require no synchronization of their own. Hints from the application as to the timing of checkpointing would likely be beneficial.

Much of the overhead imposed by checkpointing is due to the cost of copying the whole state of processes. Asynchronous and incremental checkpoints mitigate this. But asynchronous checkpointing does not decrease the load on disks and networks, and incremental checkpointing is but a conservative guess on the state that the application needs to checkpoint. Applications could be given the option to take their own checkpoints; they presumably know which portions of their state are essential, and which ones can be rebuilt from the former. The extent to which this would decrease the size of checkpoints is unclear and worth investigating.

The relaxation of application transparency amounts to customizing the fail-safety enhancements to paradigms of computation more abstract than PVM. The ultimate advantages of this can be illustrated by examples. In the bag-of-tasks paradigm a producer places tasks on a "bag" from which a collection of processes read them and complete them, returning the results to the producer. These tasks can usually be described very concisely. A fail-safe bag-of-tasks package might checkpoint only the specifications of pending tasks. The checkpoints could occur during the mutual exclusions with which the acceptance of tasks by the workers are protected. Further, incremental checkpoints would consist of a single addition or deletion to the bag. HeNCE [BDG+91] admits similar drastic optimizations. We are attempting to develop a toolkit that will hopefully be somewhat general and yet allow programmers of environments to quickly add efficient fail-safety features to their tools (e.g, to HeNCE, distributed shared objects.)

The relationship between process migration, load balancing and failures suggests more directions. Failures change the resources available to the application without changing the load. Conceptually, a new mapping from tasks to resources needs to be forged. This new mapping can be constructed by simply moving failed processes to arbitrary nodes, but there are myriad alternatives. The less rudimentary the approach, the more like a global dynamic load balancer, and the higher the cost it exerts on the recovery procedure. The first question is then how sophisticated a re-mapping is advisable after failures.

The second question is a kind of reversal on the first. Checkpoints and rollbacks implement a form of process migration that can be instantiated retroactively after a failure. Process migration is crucial for dynamic load balancing. Is it feasible and efficient to use the checkpoint/rollback mechanism as a primitive on which to implement dynamic load balancing? The answer depends on the granularity of the balancing, but work on Condor [LS92, LL90, LLM88] seems to suggest a positive answer.

The whole world cannot be checkpointed. There is a boundary between external and internal state. Internal state can be checkpointed and rolled back. External state cannot. The usual solution to this difficulty is to log all interactions across this boundary so that they can be replayed to the inside of the boundary without affecting, and without requiring control over, the outside. Other, less general, solutions may be possible. In the case of file access, for example, the state of the file system can be checkpointed incrementally by copying any currently opened files. Further optimizations are possible for applications whose files never shrink.

There are perhaps similar specific solutions for dealing with other external agents that hold part of the application's state. One such agent is the kernel, which holds information on light-weight processes, or an X server, which holds several parameters as well as the complete state of the application's display. We do not know if enough such specialized shortcuts exist, or if their assumptions are met by sufficient applications.

Last but not least, we would like to explore other mechanisms for checkpoint/rollback, including message logging/replay. We are particularly inclined to analyze and perhaps implement optimistic logging.

8. Conclusion

We have enhanced PVM so that it masks some failures from applications. We used a simple method of taking checkpoints and causing rollbacks. The performance of the system is not lacking, but further work is necessary to determine the limits of its scalability.

As a prototype, we hope this work leads towards a more general understanding of the requirements and trade-offs of fail-safety in distributed systems used for scientific computing.

9. Availability

If you are interested in trying the fail-safe PVM package, send electronic mail to Juan_Leon@cs.cmu.edu, or write to:

Juan Leon
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, Pa 15213-3891
United States of America

10. Acknowledgements

The mechanism to take a transparent checkpoint of a single process was implemented by Bennet Yee and David Applegate. Bennet is owed gratitude for that and for all the time he spent answering questions about UNIX and Mach.

We thank the PVM group for their generosity in providing a freely available and modifiable message-passing package. We particularly thank Adam Beguelin for the interest and support demonstrated through the months since his arrival at Carnegie Mellon.

References

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, Aug 1986.

[Bar81] Joel F. Bartlett. A nonstop kernel. In *Eight Symposium on Operating Systems Principles*, pages 22–29. ACM SIGOPS, Dec 1981.

[BBG+89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under unix. *ACM Transactions on Computer Systems*, 7(1):1–24, Feb 1989.

[BCZ91] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: distributed shared memory using multiprotocol release consistency. In A. Karshmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond. International Workshop Proceedings*, pages 56–60, Berlin, Germany, jul 1991. Springer-Verlag.

[BDG+91] Adam Beguelin, Jack J. Dongarra, G.A. Geist, Robert Manchek, and V.S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing '91*, pages 435–444. IEEE, Nov 1991.

[BZ91] Brian N. Bershad and Matthew J. Zekauskas. Midway: shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University. School of Computer Science, Pittsburgh, Pennsylvania, Sep 1991.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb 1985.

[DLA89] P. Dasgupta, R. LeBlanc, and W. Appelbe. The clouds distributed operating system: Functional description, implementation details and related work. In *Proceedings of the IEEE 8th International Conference on Distributed Computing Systems*, 1989.

[EJZ92] Elmootazbellah Nabil Elnohazy, David B. Johnson, and Willy Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of teh 11th Symposium on Reliable Distributed Systems*, pages 39–47, Oct 1992.

[GCCC85] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel programming in linda. In *Proceedings of IEEE International Conference on Parallel Processing*, 1985.

[GS92] G. A. Geist and V. S. Sunderam. Network based concurrent computing on the pvm system. *Journal of Concurrency: Practice and Experience*, 4(4):293–311, Jun 1992.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, (8):666–677, Aug 1978.

[ide92] No idea. No title. In *Proceedings of the International Conference on Supercomputing*, 1992. I still don't have the reference.

[KT87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, Jan 1987.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, Jul 1978.

[LB88] Pei-Jyun Leu and Bharat Bhargava. Concurrent robust checkpointing and recovery in distributed systems. In *Fourth International Conference on Data Engineering*, pages 154–163. IEEE Computer Society, IEEE Computer Society, Feb 1988.

[LL90] Mike Litzkow and Miron Livny. Experience with the condor distributed batch system. In *2nd IEEE Workshop on Experimental Distributed Systems*, pages 97–101. IEEE Computer Society, IEEE Computer Society Press, Oct 1990.

[LLM88] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - a hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111. IEEE Computer Society, Jun 1988.

[LS92] Michael Litzkow and Marvin Solomon. Supporting checkpointing and process migration outside the unix kernel. In *Usenix Winter 1992 Technical Conference*, pages 283–290, Jan 1992.

[Ran75] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, Jun 1975.

[Sun90] V.S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4), Dec 1990.

[Tse90] P.S. Tseng. A parallelizing compiler for distributed memory parallel computers. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, Jun 1990.