PROCESS CHECKPOINTING IN AN OPEN DISTRIBUTED ENVIRONMENT

Taha Osman, Andrzej Bargiela

Department of Computing
The Nottingham Trent University
Burton Street
Nottingham NG1 4BU
Email: taha@doc.ntu.ac.uk, andre@doc.ntu.ac.uk

Abstract

This paper presents a fault tolerant environment (FTE) for the reliable execution of application programs on a network of workstations. The FTE consists of an integrated error detection and fault recovery system that efficiently detects processor node crashes and hardware transient failures, and recovers user-processes from these faults using checkpointing and roll-back techniques.
We present an attractive non-blocking, user-transparent, low overhead checkpointing mechanism for roll-back recovery of failed user-application processes.

1. Introduction

The increasing complexity of real-life problems that are solved using computer technology means that computationally-intensive, time-consuming programs are an increasingly important class of applications. For instance applications that perform neural-network training, simulated annealing or programs that calculate long-term climate forecast might take anything from few hours to several days to execute [1].

Although computer systems are generally built to be reliable, the probability of hardware failure due to a human error or other external influences (e.g power supply interruption, radiation effects etc.) are non-negligible [2]. The challenge is to develop general mechanisms that monitor and optimise the use of computer systems to prevent the waste of computations accomplished prior to the occurrence of the failure.

The concept of fault-tolerant computations has originated in the context of mission critical systems where the failure to accomplish computations within stringent time constraints would lead to disastrous consequences. Fault-tolerant techniques that had been devised for such systems paid only secondary attention to the cost of providing the required reliability. Typical solutions relied on full replication of the underlying hardware, combined with software redundancy [3]. Unfortunately such approach can not be economically justified for the targeted class of applications which - although very widespread and challenging - do not have stringent real-time constraints.

The proposed fault-tolerant environment relies on recording the application process execution state at regular intervals, and restarting the process from the last recorded execution state upon hardware failure. This is known as *checkpointing* and *roll-back.*
Most of the reported research work on checkpointing have focused on the algorithms and techniques of checkpointing and its overhead on the running programs [4], [5].
In this paper, we present the design and implementation of a comprehensive framework that embraces all aspects of fault tolerance in a network-based environment. Thus we introduce a non-blocking, low overhead, user-transparent checkpointing method, as well as the mechanisms responsible for coordinating user-process backup, error detection & confinement, and process recovery in the multi-node environment.

The outline of this paper is as follows: section 2 presents error detection and recovery procedures; section 3 introduces a detailed description of the checkpointing and roll-back mechanism; experimental results and conclusions are discussed in section 4.

2. The Fault-Tolerant Environment (FTE)

2.1 Target Domain of Applications

The proposed fault-tolerant environment is designed to support a class of applications that are computationally intensive but do not have stringent real-time constrains. Therefore, they do not justify the use of expensive high-performance fault-tolerant systems, yet they necessitate the prevention of the loss of the results of the long-running computations. Example computational tasks include large scale simulations, climate forecast programs, etc. We aim to provide a *cost-effective* but *efficient* fault-tolerant solution for running these applications on network-based systems.

2.2 Design Assumptions

1) The hardware model consists of a group of Unix-based workstations connected via a communication network. One of the network nodes (usually the server) is assumed to be fault-tolerant, i.e the probability of its failure is negligible. This central host will run the main monitoring and control tasks of the FTE. The required reliability of the central host might be obtained by hardware duplication, but the detail of achieving it is beyond the scope of this paper.
2) Backup and recovery is limited to stand-alone application processes. Message-passing processes are not considered.
3) To store the processes checkpoints, a centralised NFS file system that is visible and identical to all the network hosts is assumed.
4) Failed processes can only be restarted on a host with a similar OS architecture. This is due to discrepancies in the executable file ”a.out” format and management of the process virtual address space by the MMU.

2.3 Structural Design of FTE

The context of the Fault-Tolerant environment is illustrated in figure 2.1. Upon receipt of the application-tasks specifications from the user ,the *(STARTER PROCESS)* spawns the

user tasks on the specified host(as stated in the *spawned_tasks_specs*).

Upon host crash the *(MONITOR HOST STATE)* task informs the recovery task *(RECOVER FAILED TASKS)* of the crashed host_ID so that it won't try to restart (roll-back) failed tasks on it. The failed host_ID is also sent to the *(MONITOR USER TASKS)* task to determine the IDs of the user-tasks that were running on the faulty host before the crash. These IDs are then sent to the recovery task so that it can restart the failed tasks from their most recent checkpoint.

The *(MONITOR USER TASKS)* also detects user-tasks that have exited prematurely due to a transient hardware failure and similarly sends the failed task id to the recovery task.

Task_IDs of successfully recovered user tasks are broadcasted by the recovery task so that the *(MONITOR USER TASKS)* can resume their monitoring. It's crucial that the *(MONITOR USER TASKS)* gets the ID's of the recovered tasks because they have been restarted as new executables with different task IDs.

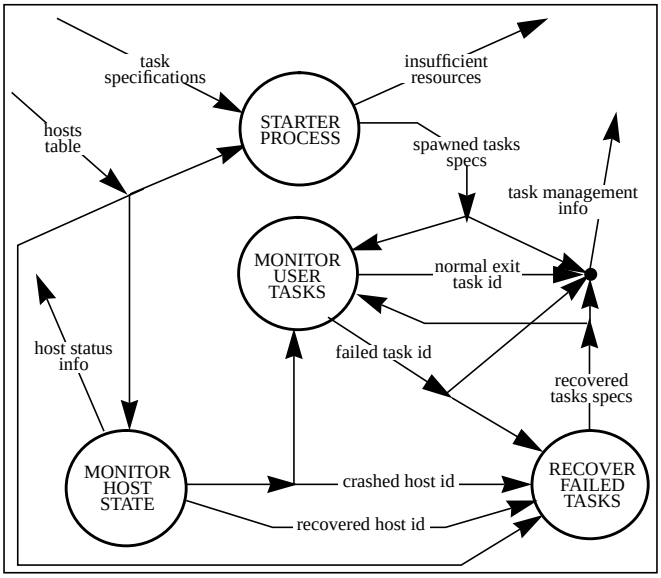

Figure 2.1 The Fault-Tolerant Environment

2.3 Detection of hardware faults

The FTE error detection mechanism detects two types of hardware faults:

1) Permanent faults: detected by the *"Monitor Host State"* task. It considers a host as crashed if it doesn't reply to acknowledgement requests within certain time *Tack* (a parameter that can be set by the user and should be less than the default give-up time of the underlying message passing interface [5]).

The crashed host is then removed from the active_host table. If the reply is received at a latter acknowledgment cycle (because of network delay) or if a message is sent by the user declaring that the host has been manually recovered, then the host is considered as "recovered", and it is added to the system host pool (see Fig 2.2).

2) Transient faults: detected initially by the OS kernel Error Detection Mechanism (EDM). Upon the detection of an error the kernel EDM kills the affected process with a signal number that corresponds to the error type.

When a user-task exits, the monitoring process *"Monitor User Tasks"* analyses the task exit_status to determine whether it exited normally or prematurely due to a failure. In both situations the task is removed from the active tasks table, but in case of task failure, the fault is reported to the task recovery mechanism.

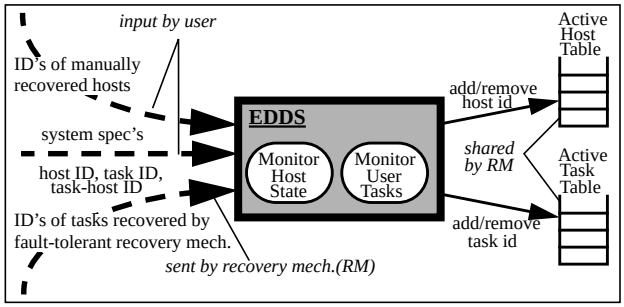

Fig. 2.2 The Error Detection Mechanism

A process-termination signal is sent to the original tasks running on the considered-as-failed host. This is necessary to prevent the duplication of the user-task. The duplication of the user-task might occur if the host's decline to reply within *Tack* is not caused by a genuine hardware failure, but by a delay in the underlying communication network, in which case we will have both the the task restarted from the last checkpoint, and the original considered-as-failed task running in FADI.

As a result of issuing the process termination signal the OS kernel will send false signals about these tasks *premature termination* upon the host's recovery from the network delay. The message originator ID has to be checked against the active tasks list to make certain that we don't restart the same task twice.

2.4 Recovery of failed user-tasks

The recovery task of the fault-tolerant distributed system acts as medium between the error detection mechanism at one end and the user-application process checkpointing and roll-back mechanism on the other. Figure 2.3 illustrates the structure diagram of the user-application process recovery task.

At the initialisation stage the *recovery* task receives the initial active host tables, then it enters an indefinite loop waiting for control messages. The *host monitor* can send two control messages: 1) informing about a host crash, in which case the host is deleted from the *active hosts table;* 2) informing about a host recovery, then the host is added to the local *active hosts table*.

It is crucial to update the *active hosts table* promptly to prevent attempts of roll-back on a faulty host and to avoid migrating the failed task into another host if the crash was caused by a network delay and the host was reinstated into the *active hosts pool.*

When the recovery task receives a message with a failed task_id it finds the CPU & OS architecture of the host on which the failed task was running and tries to restart (roll-back) the task on an active host with similar OS architec-

ture. If the restart is succesful, it passes the original checkpointing interval as an argument to the restarted task, so that it can resume checkpointing. Next the restarted task sends its new task_id to the *user-application process monitoring* task, which adds the new id to the active task table and resumes its monitoring.
In order to enhance the performance of the *error detection and recovery* algorithm, the host monitor, the user-tasks monitor and the recovery tasks were distributed into three concurrent tasks. Consequently, the user task monitor doesn't have to wait for the roll-back of a failed task to finish in order to process the exit-status of the following one, instead it notifies to the recovery task about the failed-task id and continues processingwait-exit events of other user processes
.

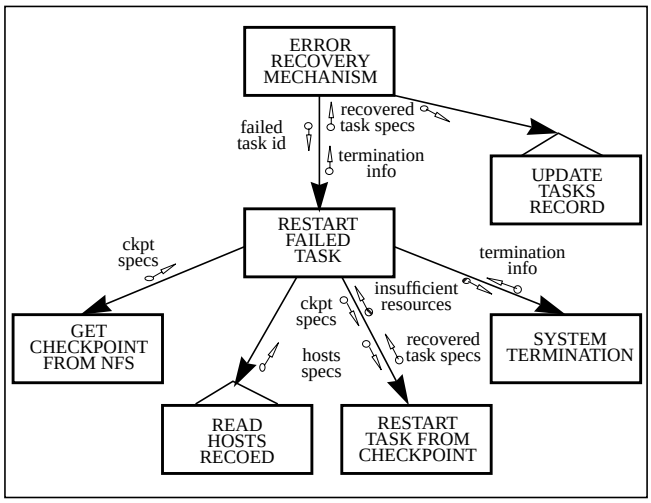

Figure 2.3 The User-Processes Recovery Task

The fact that the fault-tolerant system tasks are constructed in an event-processing fashion - that is message driven- supports acquiring more accurate approximations of the error detection latency which will be of a vital importance at the next stage of the development when we start considering the checkpointing and roll-back of interactive user-application processes.

3. The checkpointing and rollback mechanism

Checkpointing is the process of saving the local state of the process at periodic intervals, so that the process can be restarted from the last saved state (checkpoint) in case of failure. This is a cost-effective approach to fault tolerance that can be built on top of the existing computer systems.

3.1 Checkpointing techniques

From the portability viewpoint, Checkpointing techniques can be broadly categorized as:
1) Checkpointing built into the operating system. Example of such system is KeyKOS [6], checkpointing is built into the operating system kernel, it is achieved by taking frequent system-wide checkpoints of the entire system to disk. CRAY UNICOS systems [7] follows similar approach, but use selective checkpointing to create a *restart file* of any process at any time by demand.
Although usually more efficient, these checkpointing mechanisms are not portable outside their environment, and are specialized in given class of applications.
2) Checkpointing built on the top of the existing operating. The adopted checkpointing mechanism described here falls into this category.

3.1 The Checkpointing Mechanism

The basic building block of the adopted checkpointing mechanism is a technique developed by *Condor* known as *Bytestream checkpointing* [8]. The idea is that the checkpointed application process should write its state information directly into a file descriptor, and a restarting process should read that information directly from a file descriptor.
Unlike core_dump checkpointing technique suggested by the Condor group in [9], only the state information needed for process restoration has to be saved. This information is retrieved directly without the need for complex pre-processing of the OS memory maps or kernel-dump format. This increases the checkpointing process efficiency, reduces its time overhead, and supports the recovery system portability. A structured diagram of the proposed checkpointing procedure is illustrated in figure 3.1.

The checkpointing signal handler has been associated with the kernel timer mechanism. At regular intervals (ckpting interval that is determined by the user) this timer sends the signal SIGALRM to the user-program to interrupt its normal execution and invoke the checkpointing routine.
To reduce the checkpointing overhead an exact thread (child) of the user-process is created. This thread performs all the checkpoint routines, i.e reads the process state and records it into disk, while the main process (the parent) continues the execution of the application code. We call this technique *non-blocking checkpointing.*

3.2 limitations of the implemented checkpointing mechanism

1) The checkpointing mechanism can deal only with statically linked user programs. This is to ensure that it has total control over the user-process address space. Linking and loading shared libraries at execution time can disrupt the private address space thus corrupting the checkpoint image.

2) Due to the incompatibility of the UNIX systems (discrepancies in the a.out format, management of the process's address space by the MMU), rolling-back (restarting) the process from the previous checkpoint on another computer system is limited to host with similar Operating System architecture.
3) User-application programs has to be re-compiled with the FTE libraries. Despite the fact that no modifications are made to the user code, the user program must still be called as function from the checkpointing prologue to allow for initialisation and control of the checkpoint and restart of the user-program.

4. System Performance

4.1 Overview

All fault-tolerance procedures will inevitably result in a degradation in the system performance. The most significant overhead they intoduce is the overhead of actually taking

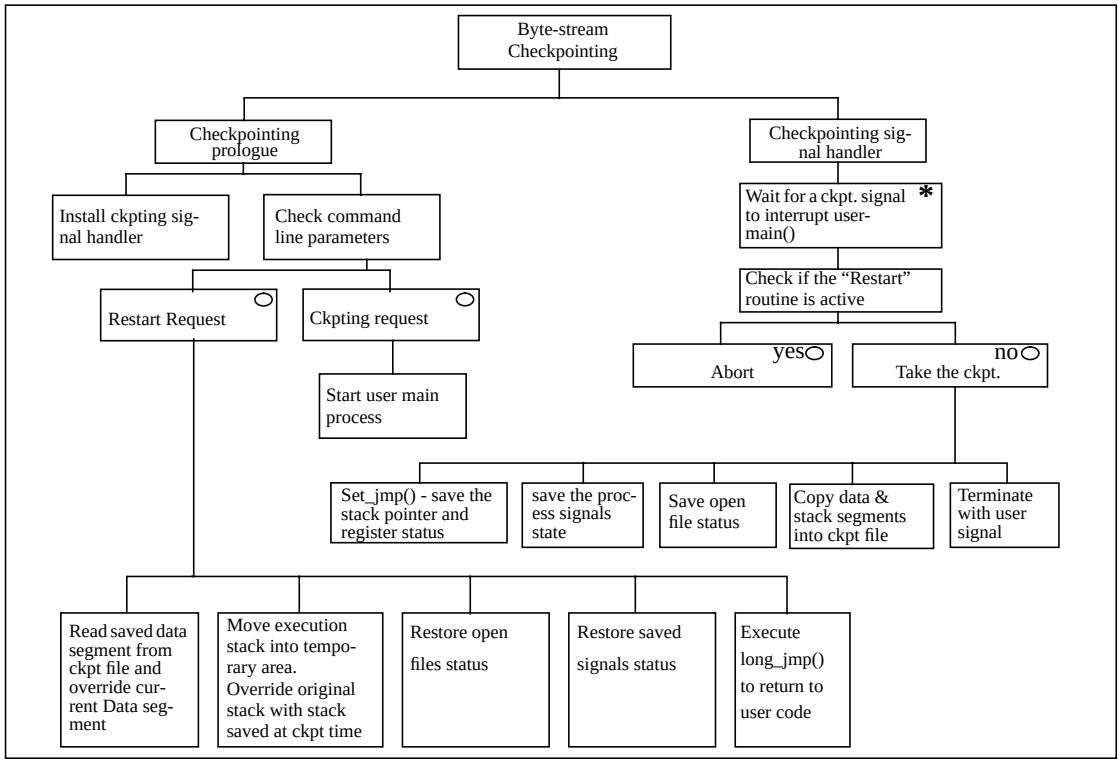

Figure 3.1 The Checkpointing Protocol

and storing the checkpoint. The non-blocking checkpointing technique that we propose in this paper reduces this overhead by interleaving the execution of the original task and the checkpointing task. Because of this concurrent execution our system affords a degree of immunity to the variation of efficeincy of taking and storing the checkpoint, as long as the current checkpointing process terminates before the next one is initiated.

The checkpoint size can increase the checkpointing overhead and occupy valuable disk space. A technique of *Incremental checkpointing* [11] has been suggested for reducing the checkpoint size and consequently the checkpointing overhead. The idea is that when a checkpoint is taken, only the portion of the checkpoint that has changed since the previous checkpoint need to be saved. The unchanged portion can be restored from previous checkpoints.
However, experimental results by *J. S. Plank and K. Li*. in *libckpt* [11] have shown that incremental checkpointing does not necessarily suit all types of applications. If large amount of the program's address space is modified between checkpoints, this can lead to little or no reduction in the checkpoint size. This conclusion is corroborated by other researchers (e.g *Pierre Sens* [12]). Consequently, we have not attempted to introduce incremental checkpointing into our procedure.

4.2 Experimental Results

The checkpointing mechanism was tested by disrupting the normal execution of the user-application process as follows: powering down the host machine, disconnecting the host machine from the ethernet, and killing the process explicitly from the command line *"kill -9"*. . Every time the user-application process was successfully rolled back from the most recent checkpoint and continued the execution on the same host or on a host with a similar OS architecture.
In order to evaluate the computational overhead of the FTE software, a numerically intensive application has been benchmarked.
The application performed an optimal decomposition of network nodes for distributed processing using the principles of *Simulated Annealing* [10]. The initial network node-configuration is read from disk and the optimization results are printed to the screen. The executables of both applications were built using GNU C++ compiler and were run on SPARC*station IPC* running SunOS 4.1.3.
Figure 4.2 shows the effect of varying the checkpointing interval from 15 to 120sec on the execution time of the simulated annealing application. It is clear that the execution time for checkpointed applications (both sequential and non-blocking) increases as we reduce the checkpointing interval.

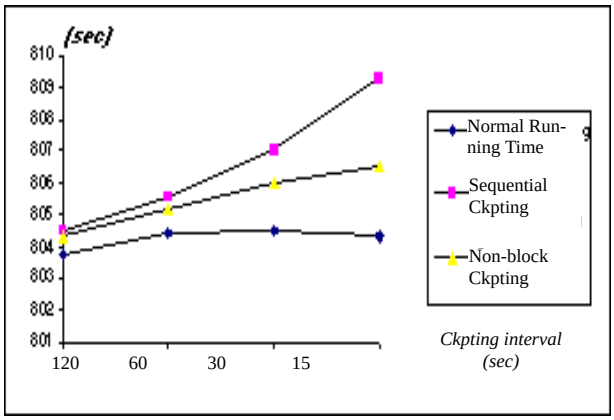

Figure 4.1 Execution Time

This ascertains the importance of balance between increasing the required level of reliability (taking checkpoints more frequently) and the subsequent degradation in performance. The task of our software is to determine the minimum possible checkpointing interval, then it is up to the user to fine-tune the reliability/performance balance.

The overhead of checkpointing is illustrated in Figure 4.2. It is obvious that non-blocking checkpointing significantly reduces the application checkpointing overhead, specifically upon short inter-checkpointing intervals (up to 300% reduction in overhead over sequential checkpointing).

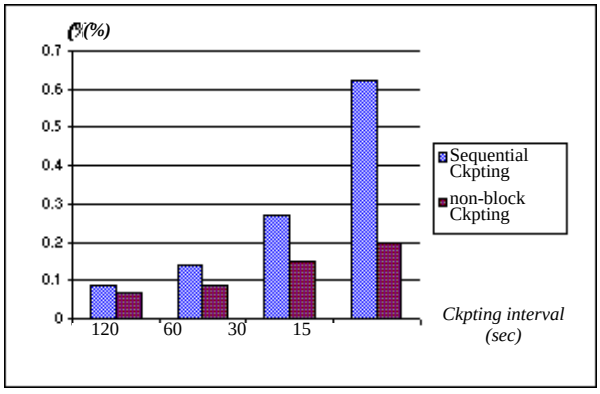

Figure 4.2 Checkpointing overhead

The results in Figure 4.2 also show that the system's performance is well bellow the rational target at 10% overhead[11], even for sequential checkpointing. The maximum recorded overhead with the minimal possible checkpointing interval (15 seconds) for Simulated Annealing using sequential checkpointing (worse possible scenario) was 0.6%.

Conclusions and Future work

This paper presented an integrative framework for fault-tolerant computing in a network-based environment.

The framework encompasses the two essential aspects of fault-tolerant processing: *detection and isolation of faults* - here we implemented an efficient mechanism to detect permanent node crashes or transient hardware failures, and *recovery of user-tasks running on the faulty hardware*- in this regard several recovery mechanisms have been considered and the *bytestream checkpointing* technique was selected as the basic building block for backup and restoration of user processes.

The checkpointing mechanism was integrated with the error detection and recovery routines to produce a reliable multi-node environment, where user-application processes can reliably execute to their logical termination despite hardware faults with graceful degradation in the system performance.

The FTE environment have satisfied the requirements of user-transparency, portability and minimal need for extra hardware support. It is especially attractive for long-running application programs, where hardware faults can waste hours or even days of useful computations.

Performance tests performed on a network of SPARC*station* IPCs running PVM daemons as the underlying communications layer, has shown that the developed error recovery mechanism has a low overhead on the running time of the tested applications. The overhead was significantly reduced even further by *forking* an exact copy of the running process to perform checkpointing without freezing the main process (non-blocking checkpointing). Performance measurements compare well with the results published of similar work in [11] and [12].

Future work will involve extending FTE to cover the backup and recovery of distributed interactive application processes.

References

[1] G.F. Coulouris and J. Dollimore. "Distributed Systems. Concepts and Design". Addison Wesley, 1993.

[2] Vounckx, G. Deconink, and others 1993. "The FTMPS-Project: Design and Implementation of Fault-Tolerance Techniques for Massively Parallel Systems". Katholieke Universitei Leuven, Belgium.

[3] B. Appel, H. Kantz. "Implications of Fault Management and Replica Determinism on the Real-Time Execution Scheme of VOTRICS"

[4] D. Pradhan, N. Vaidya. "Roll-Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture". IEEE Transactions on Computers, Vol. 43, NO. 10, Oct. 1994.

[5] Taha Osman, and Andrzej Bargiela, "Error Detection For reliable Distributed Simulations", In proceedings of the 7th European Simulation Symposium, p. 385-362, 1995.

[6] Charles Landau. "The Checkpointing Mechanism in KeyKOS". Proceedings. of the Second International Workshop on Object Orientation in Operating Systems, p. 86-89, Sept. 1992.

[7] N. Attig, V. Sander. "Automatic Checkpointing of NQS Batch Jobs on Cray Unicos". Proceedings of Cray User Group Meeting, Montreux, Spring 1993.

[8] Condor On-line Manuals - 1995. *ftp:* ftp.cs.wisc.edu

[9] Allan Briker, Michael Litzkow and Miron Livny, "CONDOR Technical Summary", University of Wisconsin - Madison, 1991.

[10] F. Hsieng Lee. "Parallel Simulated Annealing n a Message Passing Multi-Computer". PhD Thesis, Utah State University, Logan, 1995.

[11] J. S. Plank and K. Li., "A Consistent Checkpointer for Multi-computers", IEEE Parallel & Distributed Technology, 2(2):62-67, 1994.

[12] Pierre Sens and Bertil Folliot, "STAR: a Fault Tolerant System for Distributed Applications", in Proc. of the 5th IEEE Symposium on Parallel and Distributed Process ing, Dallas, Texas, pp. 656-660, Dec. 1995.