FTOP: A LIBRARY FOR FAULT TOLERANCE IN A CLUSTER

R. Badrinath *

Rakesh Gupta

Nisheeth Shrivastava

Department of Computer Science & Engineering

Indian Institute of Technology, Kharagpur

India-721302

ABSTRACT

Checkpointing and rollback recovery is a simple technique for fault tolerance. The state of a process is saved on a disk file from which the process can recover on the occurrence of failure. In this paper we describe the implementation of ***FTOP (Fault Tolerant PVM),*** a coordinated checkpointing library integrated with PVM. Existing PVM applications require only minor change for incorporating fault tolerance using FTOP. FTOP provides fault tolerance mechanism that is totally transparent to the programmer. It does not require any changes to be made in the kernel. FTOP handles intransit messages, open files and routing that makes it a very useful fault tolerant library.

KEY WORDS

Coordinated Checkpointing, Rollback recovery, Fault tolerance.

1. INTRODUCTION

Distributed systems involving Network of Workstations (***NOWs***) connected through a high speed LAN has been heavily used to meet the high computing requirements of applications. Cluster computing finds usage in scientific computation involving Finite Element method, Weather forecasting etc. NOWs are however prone to non-negligible number of faults. As the number of workstations in the cluster increases, the chances that one of them fails increases exponentially. In the event of a failure the re-execution from scratch of a long running application is undesirable and hence some kind of fault tolerance is needed.

*Currently at: INRIA/IRISA, Campus Universitaire de Beaulieu, 35000 Rennes, France. Email : rbadrina@irisa.fr

Checkpointing with rollback recovery provides an efficient mechanism for providing fault tolerance. These techniques rely on recording the application process execution state and upon failure restarting the process from the last recorded consistent state. While extensive research has been performed on checkpointing in distributed environments, there are few transparent checkpointers available to application developers[2].

The proposed system FTOP is a checkpointing library integrated with PVM (Parallel Virtual Machine. FTOP implements a transparent checkpointing technique for distributed applications running on PVM. Application developers just need to add a single statement in their PVM code and no other modification is required. Crash failures are efficiently handled without any user intervention. No changes is required in the kernel for fault tolernace. Issues such as intransit messages which arise due to assumption of reliable channels, routing of messages to migrated tasks, Open files are also taken care of.

FTOP assumes a homogeneous LINUX cluster which communicate through reliable FIFO channels. FTOP implements non blocking coordinated checkpointing and recovery algorithm [4].

This paper is divided into 7 sections. Section 2 discusses PVM. Section 3 discusses the system model and the failure model for FTOP. Section 4 discusses implementation details of FTOP checkpointing mechanism. Section 5 discusses the structure of FTOP stable storage. Section 6 discusses FTOP recovery protocol. Section 7 discusses the testing environment and the test cases.

2. PVM

PVM [8] is a library that provides a unified platform for computing over a network of heterogeneous parallel and serial computers. It allows the application designers to harness the computing power of widely available general-purpose computers (PCs), without knowing much about the configuration of the system.

The Architecture: A typical PVM system consists of a cluster of interconnected stand alone computers hosting a Global Resource manager (GRM), several PVM daemons and a runtime library called *pvmlib* linked into each application process running under it.

Every node on the cluster hosts a daemon (called PVM-daemon or pvmd), which is responsible for maintaining communication, authentication and control protocols with the virtual machine. A PVM task is a process linked with pvmlib. Similar to pvmd's they are also assigned a globally unique descriptor (tid). PVM's message passing interface uses these tid's to designate source and destination tasks for massages.

GRM is a special PVM task running on a failure free machine. It is responsible for task scheduling and coordination of checkpointing and recovery protocols. PVM provides ***reliable and FIFO communication model*** to the distributed applications. Such a message model requires the recovery procedure to handle intransit messages.

3 THE FTOP MODEL

FTOP supports fault tolerance for distributed applications using PVM. This section discusses the system and failure model assumed by FTOP.

3.1 FTOP System Model

FTOP assumes a distributed system that consists of

1. A group of LINUX based workstations connected via a high speed LAN.
2. PVM is assumed to be running on each of the workstations.
3. One of the workstations has the GRM running on it, we call this the coordinator. This node is assumed to be fault free. The required reliability of the coordinator can be achieved by hardware duplication but the detail of achieving it is beyond the scope of this paper.
4. Another workstation is configured as a stable storage. This contains the checkpoint files, the messagelogs, filelogs etc. This node is also assumed to be fault free. In fact, a single machine can be configured both as the coordinator and the stable storage.
5. All other hosts can fail.
6. The filesystem of the stable storage is NFS mounted on all the workstations. The mount point must be same on all workstations.

3.2 FTOP failure model

FTOP handles only crash failures of nodes and assumes ***fail stop model***. It does not handle process failures though the support for process failure can be easily added.

4 CHECKPOINTING

In Checkpointing, the state of a process is freezed and stored in a permanent storage, from which it can be recovered in case of any failure. FTOP is based on non-blocking coordinated checkpointing protocol [5] in which the processes orchestrate to take their checkpoints. The above was selected because it is free from Rollback propagation, Domino effect, and has less checkpointing overhead as processes can continue their computation during the checkpointing protocol [5].

4.1 Checkpointing a Linux process:

To checkpoint a process we need to dump the entire state (execution, process control and process address state) to permanent storage (usually as a disk file) in a form, which can later be reconstructed into a process.

Saving and restoring the execution state involves saving of GPRs, floating point registers, etc. which may vary depending on the architecture. Instead of going for machine dependent assembler modules for each architecture to accomplish this task, FTOP handles this through signals. The signal handling mechanism of the OS requires it to save the execution state of the process, which can later be restored once the signal has been serviced. In FTOP, a user-defined signal SIGUSR1 is send to the task at the time of checkpointing. Each task saves its execution state in the stack area, which can be restored at the time of recovery.

Text area of a process is loaded as read-only section of address space. So this section need not be checkpointed.
To checkpoint the stack area we need to know the beginning and ending address of the stack. This information is advertised by the Linux kernel through the /proc filesystem. Shared libraries are linked into the program in two stages. In compile stage only the symbols are resolved, while at the execution time the dynamic loader loads them into the process address space. There is no guarantee that libraries will be mapped into same addresses for multiple executions of the same program. So there is a possibility of shared library being mapped into different address during recovery, thus making all the dynamic links invalid. To handle this we have two options (a) update all the dynamic links at the time of recovery or (b) we have to ensure that the libraries are mapped to the same address on recovery. FTOP implements the second one.

4.2 Checkpointing a distributed application:

As mentioned earlier, FTOP implements non blocking coordinated checkpointing protocol.

The GRM initiates the checkpointing protocol on receiving the SIGALRM signal. It sends an initiate message (SM_CKPTSIGNAL) to all daemons in the virtual machine. On receiving the initiate message the daemons start the local checkpointing process by asking (SIGUSR1) each task under them to take their local checkpoints. Tasks after taking checkpoint send an ack (TM_CKPTDONE) back to the daemon, which on receiving acks from all the tasks sends an ack (SM_CKPTDONE) back to the GRM. The GRM on receiving ack messages from all the daemons commits (SM_CKPTCOMMIT & SIGUSR1) the checkpoint and inform each task about the successful completion of the protocol through their daemons.

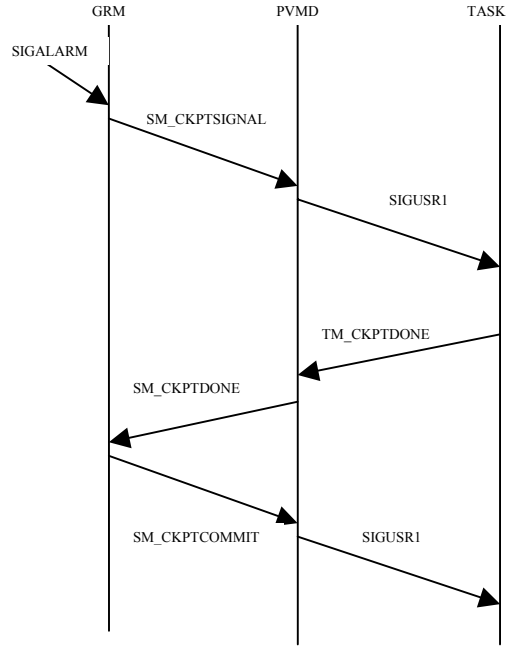

Figure 1 Timing Diagram for Checkpointing protocol

Note that FTOP implements non blocking checkpoint protocol in which the tasks continue with their execution once they have taken their checkpoints. For this two more protocol messages are required to add consistency (fig 2).

TM_CKPTSIGNAL: This message is sent by the task to its daemon before each application message when the checkpoint protocol is in progress. If the application message is destined for a local task the daemon finds the status of the destination. If the destination has taken checkpoint then sends the application message to the task otherwise it waits for the task to take checkpoint before delivering the application message.

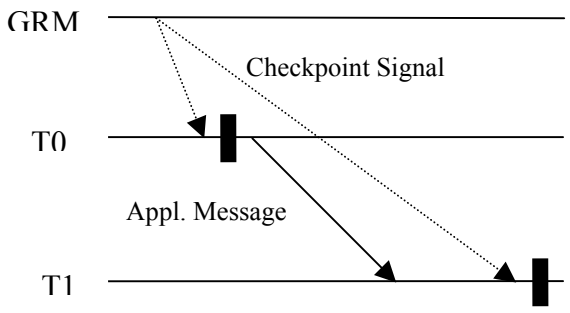

Figure 2 Problem in Non-Blocking Coordinated Checkpointing Protocol

If the application message is destined to a foreign task the daemon sends the DM_CKPTSIGNAL to the destination daemon before routing the message.

DM_CKPTSIGNAL: This is send by a daemon to another daemon as discussed above. If checkpointing has not been initiated in the destination host the destination daemon starts the checkpointing protocol and ensures that the destination tasks have taken checkpoint before forwarding the message.

We claim that the protocol ***always takes a consistent checkpoint.*** A checkpoint is inconsistent if the receive event of a message is checkpointed but the send event is not. For consistency all the messages send after a particular checkpoint should be included in the next (or later) checkpoint image of the receiver. TM_CKPTSIGNAL and DM_ CKPTSIGNAL ensure this.

4.3 INTRANSIT Messages

Intransit messages are defined as those messages whose send has been recorded but the receive has not been recorded. The assumption of reliable communication channel leads to the requirement of checkpointing communication channels between the processes.

The basic idea is to log the sends and receives of all the messages to stable storage at the time of checkpointing. When failure occurs, the logs can be examined to determine which messages were in transit at the time of checkpoint and can be replayed. A lot of issues arise in logging and replaying of these messages. We discuss them briefly.

- Some stub function for sending and receiving messages is needed. This will provide the sequence number for all sends and receive events.
- A directory structure using which the process can find and replay the intransit messages.
- A simple garbage collector.

4.4 Open Files

FTOP handles files in an efficient manner, though it handles only file reads and file appends. Arbitrary file writes are not handled because they are not quite common and involve a lot of overhead. FTOP has the following underlying assumptions for handling files

- Files are accessed with the same name on all nodes.
- Files are opened for read and append only.

To checkpoint the state of files the name of the files, their descriptors, mode of opening and the read write pointers need to be known. FTOP extracts all this information by using the LINUX utility ***'lsof'.*** The read write pointers are extracted using ***lseek*** call on the open files. These information about the open files are kept in the filelog corresponding to the task in the stable storage.

5 THE STABLE STORAGE:

FTOP requires the stable storage to be a failure free machine that must be mounted on same location on all the NOWs. It is used to store the checkpointed images and message logs.

The process image is saved in Ckpt directory. Although coordinated checkpointing requires only the most recent checkpoint for recovery, two files for each process are needed [5]. Similar is the case of file logs (fileinfo). Hence two log files are required for storing file information.

6 RECOVERY

Recovery involves fault detection, fault assessment and fault recovery. In FTOP fault detection mechanism comes in inbuilt with PVM. An idle PVM daemon occasionally checks/polls its peers by continuously sending ping packets. When a PVM daemon times out while communicating with its peer, it informs the other live daemons and the GRM about the failed host.

The coordinator i.e. the GRM performs fault assessment. The GRM identifies the failed host and the tasks running on them. It handles issues of tasks making a normal exit and tasks that were spawned after the recovery line. The tasks that made a normal exit after recovery line need to be recovered at the time of recovery. The tasks that were spawned after the last consistent set of checkpoints need to be killed. The failed tasks are spawned on appropriate host depending upon the load on each.

Fault recovery involves the restoration of the failure free state of execution. A 2 phase blocking protocol is implemented in FTOP for fault recovery. In the first phase the GRM informs each of the tasks to roll back to their last committed checkpoint. When all the tasks have successfully rolled back to their last consistent checkpoint the GRM sends a commit message to the task. The actual protocol messages are discussed in figure 3. The tasks are not allowed to send and receive messages when the recovery protocol is in progress.

6.1 Restoration of the Local state of a linux process

The processor context saved due to the invocation of the last checkpoint signal handler contains information regarding where execution should resume in the user's code once failure is detected. On the occurrence of failure the process after restoring its address space must jump to the point of execution where the execution of the last checkpointing had completed. For this FTOP uses the setjmp( ) and longjmp( ) system calls (see man pages on LINUX for details). setjmp is called with the user defined buffer JMP_BUF as argument. This saves the PC value, the stack pointer, base pointer, and other state information. If somewhere in the program longjmp( ) is called with the JMP_BUF as argument than the PC value and other state information restores to the value which was there when the original setjmp() call was made. This mechanism is used for restoration of the execution state. During checkpoint the process calls the setjmp function and than takes the checkpoint and returns from the checkpoint signal handler. At the time of recovery the process after restoring the address space calls longjmp( ) with the same JMP_BUF. This takes the program to the point where the checkpoint was taken. After returning from the checkpoint handler the execution state of the process prior to checkpoint is restored.

6.2 Process address space

The FTOP does not restore the read only part of the process address space. It only restores the writeable and the shareable parts of the process address space.

As discussed in section 4, the shared libraries should be mapped at exactly the same location where they were mapped during the failure free execution to prevent the dynamic links from becoming stale. In FTOP dynamic libraries are restored by first creating the mapped segment in the virtual address space using the mmap( ) function call. The protection and attribute flags for this new segment are set to those saved in the checkpoint file except that the segment always has write access and the memory is always marked as private. Write access is necessary so that the saved bytes in the checkpoint file can be written to the segment.

The most difficult part to restore is the stack area of the process address space. The problem is that the saved stack information may overwrite the call frame of the procedure doing restart. In FTOP Whenever the checkpointed stack need to be restored, a check of whether the current stack frame is above the old stack or not is done. So if the current stack frame is within the range of the saved stack the same function is called recursively until the current stack frame is below the old stack. Then the contents of

saved stack are copied from file to the virtual memory space of the process.

6.3 Communication channel

To restore the state of the communication channel FTOP replays all intransit messages. Each process after restoring its execution state and address space looks into the log to find out all intransit messages. A sequence number references each message from a particular source to a particular destination. The process finds the sequence number of the last message send by it to a destination and the sequence number of the last message received by the destination from it. All messages whose sequence number lies between these two messages are intransit messages and need to be replayed.

6.4 Open files

There are several methods for checkpointing open files, which are discussed in the literature. This includes Shadow copy (Libckpt[3]), in place update with undo logs (*winckp [*5], and *SCR* algorithm [6]), modification operation buffering (7).

Since we are not using random read/write on files, these techniques introduce a lot of overhead, so FTOP incorporates a novel technique that has very little overhead in handling files. Restoring the open files involves

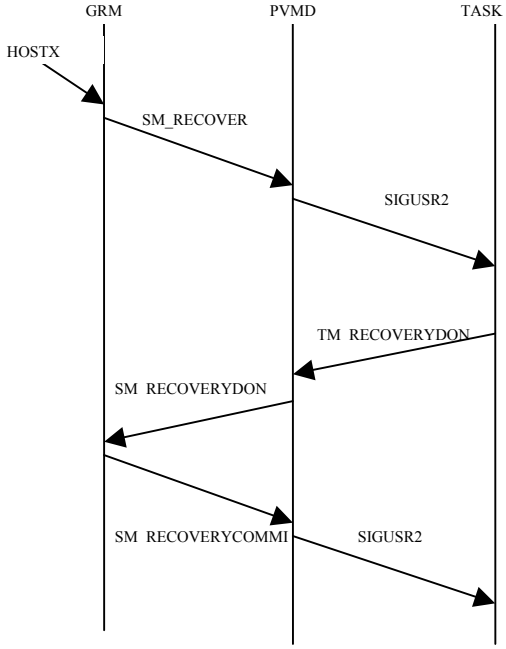

Figure 3 Timing Diagram for Recovery Protocol

reading of the file log for the process to determine the state of the files opened by the process until the last committed checkpoint. In FTOP once this information is got each of the files are reopened and their descriptors are duped to the new descriptor. The read write pointers are restored by seeking to the read write position as kept in the checkpoint file.

6.5 Message Routing:

In PVM, message routing is done according to the task-id (tid). During recovery, some tasks may be re-spawned (probably on a different host) and given a new tid. But since in FTOP, fault tolerance is transparent, a task, which wants to send message to the respawned task will never know about the failure and will continue to send messages addressed to the old tid. This gives rise to the problem of routing. To handle this, FTOP maintains a mapping between the oldest tid (a task may fail multiple number of times) of the task and its current tid. This mapping exists in the form of a route-table in every daemon. Whenever a message destined to a non-existent tid arrives the daemon scans the route table to find the corresponding mapping, and routes the packet appropriately.

7 TESTING:

Various applications such as Matrix multiplication and POVRAY were tested on FTOP.

7.1 Testing Environment:

We have tried to emulate the cluster environment as close as possible. The system contains a network of workstations and a machine configured as a stable storage. The configuration was:

- Pentium III Workstations with Red Hat Linux 7.1, connected through 100 Mbps Ethernet LAN serve as the nodes in the cluster.
- One of the workstations is configured to be the stable storage. The Stable storage is NFS mounted on a specific directory on each of the nodes.
- The application chosen to measure the performance was straightforward matrix multiplication. Two 700*700 matrix were randomly generated and were multiplied. The variation of the execution time with the checkpointing interval is reported in the following graph.

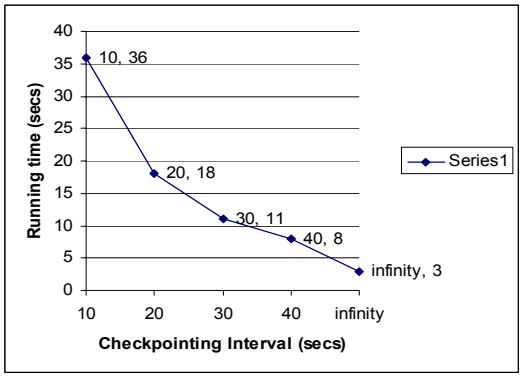

Figure 4 Variation of Exec time with checkpointing interval

- The other application that was used to measure the performance was *PVMPOV*. It is a full featured distributed ray tracer build on PVM.

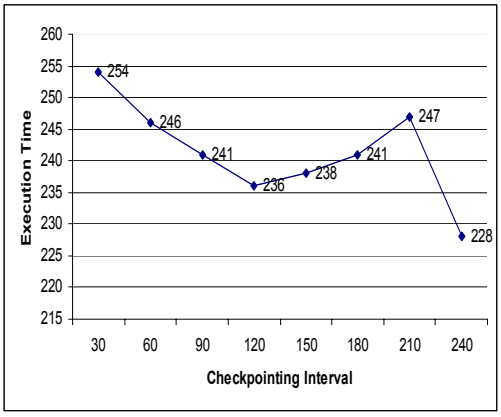

Figure 5 Performance results for PVMPOV

Interval	No. of checkpoints	Execution time
30	7	254
60	3	246
90	2	241
120	1	236
150	1	238
180	1	241
210	1	247
240	0	228

Since the checkpoint overhead is inversely related to the checkpointing interval therefore the downward slope is seen in the initial phase of the graph. However there is an increase in the overhead in later phase due to increase in checkpoint size arising due to the increase in number of messages to be logged. The last point provides the execution time without any checkpoint session.

8 CONCLUSION

This paper describes an effort to build a fault tolerance into the standard PVM model staying entirely at the user level. The methods are such that they can be carried over to other similar environments. While we have been able to rollback the open files state of the process, other state such as device association may require explicit OS support. As a future direction we intend to integrate well known optimizations into the checkpointing protocols, and aim to support checkpointing schemes other than the coordinated checkpointing explored in this paper.

REFERENCES

[1] Taha Osman and Andrez Bargiela. Process Checkpointing in an open distributed environment., *Proceedings of European Simulation Multiconference, ESM'97,* Istambul, June 1997, 536-540 .

[2] Yuqun Chen, James S. Plank and Kai Li. CLIP: A Checkpoint tool for message passing parallel programs. *Proceedings of Supercomputer '97* , San Jose , California, November 1997.

[3] James S. Plank, Micah Beck and Gerry Kingsley. Libckpt: Transparent Checkpointing under Unix. *Proceedings of the Usenix Winter Technical Conference,* New Orleans, LA, January 1995, 213-223

[4] Y-M. Wang, Y. Huang, K-P Vo, P-Y Chung and C. Kintala, Checkpointing and its Applications, *Proceedings of the 25th Symposium on Fault-Tolerant Computing*, June 1995, pp. 22-31.

[5] P.E. Chung, W-J Lee, Y. Huang, D. Liang and C-Y Wang, Winckp: A Transparent Checkpointing and Rollback Recovery Tool for Windows NT applications, *Proceedings of IEEE 29th Symposium on Fault- Tolerant Computing*, Madison, June 1999, 220-223 .

[6] Wei Xiao-Hui and Ju Jiu-Bin. SCR algorithm: saving/restoring states of file systems. *Operating Systems/Review*, 33(1), Jan. 1999, 26-33.

[7] Dan Pei, Modification Operations Buffering: A Low-overhead Approach to Checkpoint User Files, *Proceedings of IEEE 29th Symposium on Fault-Tolerant Computing (Student paper),* Madison, June 1999, 36-38.

[8] J. J. Dongarra, A. Geist, R.J. Manchek, and V.S. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics:7(2)*, April 1993, 166-175.