Bhushan Sapre, Anup Garje, Dr. B. B. Mesharm/ International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622 www.ijera.com
VOL. 1, ISSUE 3, PP.1065-1071

Fault Tolerant Environment Using Hardware Failure Detection, Roll Forward Recovery Approach and Microrebooting For Distributed Systems

Bhushan Sapre*, Anup Garje**, Dr. B. B. Mesharm***

*(Department of Computer Technology, Veetmata Jijabai Technilogical Institute, Matunga, Mumbai)
**(Department of Computer Technology, Veetmata Jijabai Technilogical Institute, Matunga, Mumbai)
***(Head of Dept. of Computer Technology, Veermata Jijabai Technological Institute, Matunga Mumbai)

ABSTRACT

Fault Tolerant Environment is a complete programming environment for the reliable execution of distributed application programs. Fault Tolerant Distributed Environment encompasses all aspects of modern fault-tolerant distributed computing. The built-in user-transparent error detection mechanism covers processor node crashes and hardware transient failures. The mechanism also integrates user-assisted error checks into the system failure model. The nucleus non-blocking checkpointing mechanism combined with a novel low overhead roll forward recovery scheme delivers an efficient, low-overload backup and recovery mechanism for distributed processes. Fault Tolerant Distributed Environment also provides a means of remote automatic process allocation on distributed system nodes. In case of recovery is not possible, we can use new microrebooting approach to store the system to stable state.

1. INTRODUCTION

Though cloud computing is rapidly developing field, it is generally accepted that distributed systems represent the backbone of today's computing world. One of their obvious benefits is that distributed systems possess the ability to solve complex computational problems requiring large computational by dividing them into smaller problems. Distributed systems help to exploit parallelism to speed-up execution of computation-hungry applications such as neural-network training or various system modeling. Another benefit of distributed systems is that they reflect the global business and social environment in which we live and work. The implementation of electronic commerce, flight reservation systems, satellite surveillance systems or real-time telemetry systems is unthinkable without the services or intra and global distributed systems.

These areas require that their downtime is negligible. The deployment of distributed systems in these areas has put extra demand on their reliability and availability. In a distributed system that is running number applications it is important to provide fault-tolerance, to avoid the waste of computations accomplished on the whole distributed system when one of its nodes fails to ensure failure transparency. Consistency is also one of the measure requirements. In on-line and mission-critical systems a fault in the system operation can disrupt control systems, hurt sales, or endanger human lives. Distributed environments running such applications must be highly available, i.e. they should continue to provide stable and accurate services despite faults in the underlying hardware.

Fault tolerant environment was developed to harness the computational power of interconnected workstations to deliver reliable distributed computing services in the presence of hardware faults affecting individual nodes in a distributed system. To achieve the stated objective, Fault tolerant environment has to support the autonomic distribution or the application processes and provide means for user-transparent fault-tolerance in a multi-node environment[10].

Addressing the reliability issues of distributed systems involves tackling two problems: error detection and process recovery. Error detection is concerned with permanent and transient computer hardware faults as well as faults in the application software and the communication links. The recovery of failed distributed applications requires recovering the local execution state of the processes, as well as taking into consideration the state of the communication channels between them at the time of failure[10].

2 RELATED WORK

Fault-tolerance methods for distributed systems have developed in two streams: checkpointing/rollback recovery and process-replication mechanisms.

Process replication techniques have been widely

studied by many researchers. In this technique, required processes are replicated and executed on different machines. The assumption is made that all replicas of same process will not fail at the same point of time and an unfailed replica can be used to recover other replicas. Although these techniques incur a smaller degradation in performance when compared to checkpointing mechanisms, they are not overhead-free. Updating of one replica requires that the other replicas must be updated to maintain consistency. However, the main hindrance to wide adoption of process-replication methods in various areas is the heavy cost of the redundant hardware needed for the execution of the replicas.

In contrast with process replication mechanisms, the other technique does not require duplication of hardware or replication of processes. Instead, each process periodically records its current state and/or some history of the system in stable storage, and this action called *checkpointing*. If a failure occurs, processes return to the previous checkpoint *(rollback)* and resume their execution from this checkpoint. A checkpoint is a snapshot of the local state of a process along timeline, saved on local non-volatile storage to survive process failures. A global checkpoint of an n-process distributed system consists of n checkpoints (local) such that each of these n checkpoints corresponds uniquely to one of the n processes. A global checkpoint C is defined as a consistent global checkpoint if no message is sent after a checkpoint of C and received before another checkpoint of C. The checkpoints belonging to a consistent global checkpoint are called globally consistent checkpoints (GCCs).The overhead that this technique incurs is greater than that of process replication mechanisms because checkpoints are taken during failure-free operation of processes, and rollback-recovery requires certain actions to be performed to ensure consistency of system when processes recover from crash.

The concept of roll-forward checkpointing is considered to achieve a simple recovery comparable to that in the synchronous approach. This concept helps in limiting the amount of rollback of a process (known as domino effect) in the event of a failure. The roll-forward checkpointing approach has been chosen as the basis of the because of its simplicity and some important advantages it offers from the viewpoints of both checkpointing and recovery.

In case the recovery using r0ll-forward approach is not possible, then we have no choice other than restoring

the system to a best known stable state. To restore the system to a best known stable state, the best option is rebooting the system. Rebooting involves restarting of all the components of the system including those once which were working correctly before the system failed. Restarting the whole system can be time consuming sometime which increases the downtime of the system. This is not tolerable in the distributed systems that work 24/7. Therefore we can minimize the rebooting using microreboot approach. In this approach, only those components are rebooted which failed in the system.

2.1 Work done in Roll-forward checkpointing

The objective of the algorithm is to design a checkpointing / recovery algorithm that will limit the effect of the domino phenomenon in a distributed computation while at the same time will offer a recovery mechanism that is as simple as in the synchronous checkpointing approach. In order to achieve its objective, processes go on taking checkpoints (basic checkpoints) asynchronously whenever needed whereas the roll-forward checkpointing algorithm runs periodically (say the time period is T) by an initiator process to determine the GCCs. During the execution of the algorithm an application process P is forced to take a checkpoint if it has sent an application message m after its latest basic checkpoint which was taken by process asynchronously. It means that the message m cannot remain an orphan because of the presence of the forced checkpoint because every sent message is recorded. It implies that in the event of a failure occurring in the distributed system before the next periodic execution of the algorithm, process P can restart simply from this forced checkpoint after the system recovers from the failure. However, if process P has not sent any message after its latest basic checkpoint, the algorithm does not force the process to take a checkpoint. In such a situation process P can restart simply from its latest basic checkpoint[7].

2.2 Difference between Roll forward and Rollback approach

- Roll forward stores only latest checkpoint and rollback stores all checkpoints and requires truncation.
- Roll forward takes two kinds of checkpoints forced and co-ordinated while roll back takes only coordinated
- In roll forward, every process takes forced checkpoint after it sends message, it is not necessary in rollback approach
- Roll forward guarantees that no orphans exist while roll back makes no such promise[5].

3 PROPOSED SYSTEM

3.1 Overview of the Fault tolerant environment

Fault tolerant environment is designed to support computation-intensive applications executing on networked workstations such as Railway reservation system. While the 'economy of effort' stipulates the introduction of measures that prevent the loss of the long-running computation results executing on distributed nodes, frequently it is difficult to justify the use or expansive hardware replication fault-tolerance techniques[10]. Another relevant area for the application of Fault tolerant environment is on-line distributed programs such as decision support control mechanisms and flight reservation systems. Such systems can tolerate a short delay in services -for fault-management, but a complete halt of the system upon the occurrence of faults cannot be accepted i.e. they require immediate recovery for the failure.

Figure presents an abstract view of Fault tolerant environment operation. The error detection and fault recovery modules run on a central node (server) that is assumed to be fault-tolerant, i.e. the probability of its failure is negligible. The required reliability of the central node might be obtained by hardware duplication. It is assumed that the probability of the hardware failure of central server is negligible. Upon system start-up, Fault tolerant environment identities the configuration of the underlying network and presents it to the user, which selects the network node(s) on which the application processes should be executed. Fault tolerant environment spawns the application processes on the specified nodes and periodically triggers their checkpointing to save their execution image together with any inter-process messages which are saved into stable storage Nodes participating in the application execution are continuously monitored, and in the event of no& crash, checkpoints of all the processes running on the failed node arc extracted from stable storage and the processes arc restarted on an operative node.

3.2 Detection of faults in the hardware environment

The starting point for all fault-tolerant strategies is the detection of an erroneous state that, in the absence of any corrective actions, could lead to a failure of the system. Fault tolerant environment error detection mechanism (EDM) identifies two types of hardware faults: processor node crashes (as caused by power failure) and transient hardware failures (temporary memory flips, bus errors, etc.) that cause the failure of a single application process, and also allows the integration of user-programmed (application-specific) error checks[10].

3.3 Detecting node failures

Detection of node failures is based on a central node monitoring task that periodically sends acknowledgement requests to all the nodes in the system.

$$RTT_{i+1} = \alpha \times RTT_i + (1 - \alpha) \times Srtt$$

Each node must acknowledge within a predefined time interval (acknowledgment timeout), otherwise it will be considered as having 'crashed'.

The Acknowledgement timeout is calculated as

$$t_{latency} = E(t_{edm}) + 3 \times \sigma(t_{edm}) + \frac{rtt}{2}$$

Where

- RTT_i is the current estimate of round-trip time
- RTT_{i+1}, is the new computed value, and
- α is a constant between 0 and 1 that controls how rapidly the estimated *rtt* adapts to the change in network load

3.2 Detecting application-process failures

When a user-process exits, analyses the process exit status is analyzed by Fault tolerant environment to determine whether it exited normally or prematurely due to a failure, in which case the failed process recovery is initiated. With regard to the user-assisted error detection, a special signal handler was dedicated to service the detection of such errors. All the programmer has to do is to raise an interrupt with a predefined signal number and the detection mechanism will handle the error as if it was raised by the kerne1 (OS) detection mechanism (KDM).

For a centralized detection mechanism - such as Fault tolerant environment's, it is vital to consider the

latency or detecting errors on the distributed system nodes.

3.3 Recovery of distributed application processes

3.3.1 Creation of checkpoints

Assume that the distributed system has n processes (P_0, P_1, . . . , P_i, . . . , P_{n-1}). Let C^x_i ($0 \leq i \leq n - 1$, $x > 0$) denote the x-th checkpoint of process P_i, where i is the process identifier, and x is the checkpoint number. Each process P_i maintains a flag c_i (Boolean). The flag is initially set at zero. It is set at 1 only when process P_i sends its first application message after its latest checkpoint. It is reset to 0 again after process P_i takes a checkpoint. Flag c_i is stored in local RAM of the processor running process P_i for its faster updating. Note that the flag c_i is set to 1 only once independent of how many messages process P_i sends after its latest checkpoint. In addition, process Pi maintains an integer variable N_i which is initially set at 0 and is incremented by 1 each time the algorithm is invoked. As in the classical synchronous approach, we assume that besides the system of n application processes, there exists an initiator process P_I that invokes the execution of the algorithm to determine the GCCs periodically. However, we have shown later that the proposed algorithm can easily be modified so that the application processes can assume the role of the initiator process in turn. We assume that a checkpoint C^x_i will be stored in stable storage if it is a GCC; otherwise in the disk unit of the processor running the process P_i replacing its previous checkpoint C^{x-1}_i. We have shown that the proposed algorithm considers only the recent checkpoints of the processes to determine a consistent global checkpoint of the system. We assume that the initiator process P_I broadcasts a control message M_{ask} to all processes asking them to take their respective checkpoints. The time between successive invocations of the algorithm is assumed to be much larger than the individual time periods of the application processes used to take their basic checkpoints.

In this work, unless otherwise specified by 'a process' we mean an application (computing) process.

Example 1: Consider the system shown in Fig. 1. Examine the diagram (left of the dotted line). At the starting states of the processes P_0 and P_1, the flags c_0 and c_1 are initialised to zero. The flag c_1 is set at 1 when process P_1 decides to send the message m_1 to P_0. It is reset to 0 when process P_1 takes its basic checkpoint C_1^1. Observe that the flag c_1 is set to 1 only once irrespective of how many messages process P_1 has sent before taking the checkpoint C_1^1. Process P_1 has not sent any message between checkpoints C_1^1 and C_1^2. So, c_1 remains at 0. Also it is clear why c_1 still remains at 0 after the checkpoint C_1^2. Process P_0 sets its flag c_0 to 1 when it decides to send the message m_3 after its latest checkpoint C_0^1 [9].

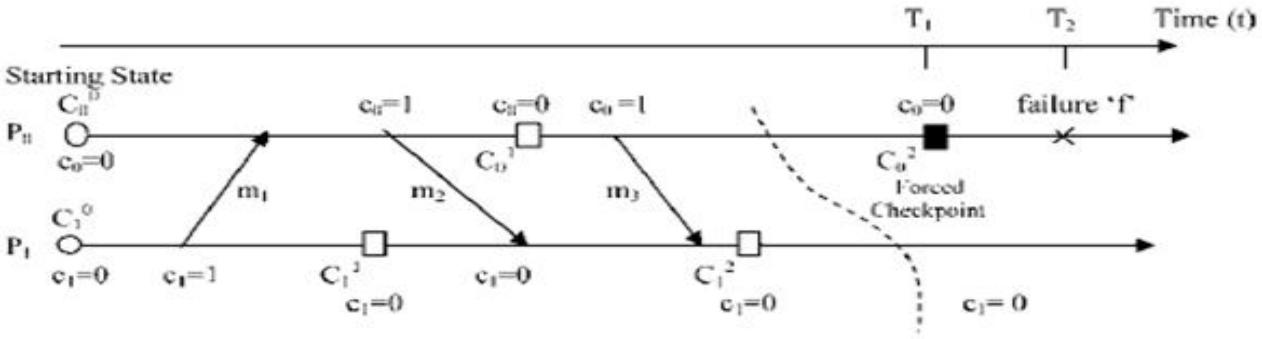

Fig.2 Updating of flags c_0 and c_1

3.3.2 Significance of forced checkpoints

Consider the system of Fig. 1 (ignore the checkpoint C_2^0 for the time being). Suppose at time T_2 a failure 'f' occurs. According to the asynchronous approach processes P_0 and P_1 will restart their computation from C_0^1 and C_1^1, since these are the latest GCCs.

Now, consider a different approach. Suppose, at time T_1, an attempt is made to determine the GCCs using the idea of forced checkpoints. We start with the recent checkpoints C_0^1 and C_1^2 , and find that the message m_3 is an orphan. Observe that the flag c_o of process P_0 is 1, which means that process P_0 has not yet taken a checkpoint after sending the message m_3. However, if at time T_1 process P_0 is forced to take the checkpoint C_0^2 (which is not a basic checkpoint of P_0), this newly created checkpoint C_0^2 becomes consistent with C_1^2 . Now, if a failure 'f' occurs at time T_2, then after recovery, P_0 and P_1 can simply restart their computation from their respective consistent states C_0^2 and C_1^2 . Observe that process P_1 now restarts from C_1^2 in

the new situation instead of restarting from C_1^1. Therefore the amount of rollback per process has been reduced. Note that these two latest checkpoints form a recent consistent global checkpoint as in the synchronous approach. The following condition states when a process has to take a forced checkpoint[9].

Condition C: For a given set of the latest checkpoints (basic), each from a different process in a distributed system, a process P_i is forced to take a checkpoint C_i^{m+1}, if after its previous checkpoint C_i^m belonging to the set, its flag $c_i = 1$.

3.3.3 Non-blocking approach

We explain first the problem associated with non-blocking approach. Consider a system of two processes P_i and P_j. Assume that both processes have sent messages after their last checkpoints. So both c_i and c_j are set at 1. Assume that the initiator process P_I has sent the request message Mask. Let the request reach P_i before P_j. Then P_i takes its checkpoint C_i^k because $c_i = 1$ and sends a message m_i to P_j. Now consider the following scenario. Suppose a little later process P_j receives m_i and still P_j has not received M_{ask}. So, P_j processes the message. Now the request from P_I arrives at P_j. Process P_j finds that $c_j = 1$. So it takes a checkpoint C_j^r. We find that message mi has become an orphan because of the checkpoint C_j^r. Hence, C_i^k and C_j^r cannot be consistent[9].

3.3.4 Solution to the non-blocking problem

To solve this problem, we propose that a process be allowed to send both piggybacked and non-piggybacked application messages. We explain the idea below. Each process P_i maintains an integer variable N_i, initially set at 0 and is incremented by 1 each time process P_i receives the message Mask from the initiator. Thus variable N_i represents how many times the check pointing algorithm has been executed including the current one (according to the knowledge of process P_i). Note that at any given time t, for any two processes P_i and P_j, their corresponding variables N_i and N_j may not have the same values. It depends on which process has received the message Mask first. However, it is obvious that $|N_i - N_j|$ is either 0 or 1. Below we first state the solution for a two-process system. The idea is similarly applicable for an n process system as well.

Two-process solution:

Consider a distributed system of two processes P_i and P_j only. Assume that P_i has received Mask from the initiator process P_I for the k th execution of the algorithm,

and has taken a decision whether to take a checkpoint or not, and then has implemented its decision. Also assume that P_i now wants to send an application message m_i for the first time to P_j after it finished participating in the k th execution of the algorithm. Observe that P_i has no idea whether P_j has received Mask yet and has taken its checkpoint. To make sure that the message m_i can never be an orphan, P_i piggybacks m_i with the variable N_i. Process P_j receives the piggybacked message $<m_i, N_i>$ from P_i. We now explain why message m_i can never been an orphan. Note that $N_i = k$; that is it is the kth execution of the algorithm that process P_i has last been involved with. It means the following to the receiver P_j of this piggybacked message:

(1) Process P_i has already received Mask from the initiator P_I for the k th execution of the algorithm,

(2) P_i has taken a decision if it needs to take a forced checkpoint and has implemented it,

(3) P_i has resumed its normal operation and then has sent this application message m_i.

(4) The sending event of message mi has not yet been recorded by P_i.

3.4 Microrebooting

Microrebooting in a new approach for restoring the system to a stable state. Rebooting is generally accepted as a universal form of recovery for many software failures in the industry, even when the exact causes of failure are unknown. Rebooting provides a high-confidence way to reclaim stale or leaked resources. Rebooting is easy to perform and automate and it returns the system to the best known, understood and tested state[8].

Unfortunately rebooting for an unexpected crash can take a very long time to reconstruct the state. A microreboot is the selective crash-restart of only those parts of a system that trigger the observed failure. This technique aims to preserve the recovery advantages of rebooting while mitigating the drawbacks. In general, a small subset of components is often responsible for a global system failure, thus making the microreboot an effective technique for system-global recovery.

By reducing the recovery to the smaller subset of components, microrebooting minimizes the amount of state loss and reconstruction. To reduce the state loss, we need to store the state that must survive the microrebooting process into separate repositories which are crash safe. This separates the problem of data recovery from application-

logic recovery and lets us perform the latter at finer grain than the process level.

Microreboots are largely as effective as full reboots but 30 times faster. In our prototype, microreboots recover from a large category of failures for which system administrators normally restart the application, including deadlocked or hung threads, memory leaks, and corrupt volatile data. If a component microreboot doesn't correct the failure, we can progressively restart larger subsets of components.

Because the component-level reboot time is determined by how long the system takes to restart the component and the component takes to reinitialize, a microrebootable application should aim for components that are as small as possible[8].

Microrebooting just the necessary components reduces not only recovery time but also its effects on the system's end users.

4 CONCLUSION

Fault tolerant environment was designed to provide a fault-tolerant distributed environment that provides distributed system users and parallel programmers with an integrated processing environment, where they can reliably execute their concurrent(distributed) applications despite errors that might occur in the underlying hardware.

Fault tolerant environment user-transparent error detection mechanism covers processor node crashes and hardware transient failures, and allows for the integration of user-programmed error checks into the detectable errors database.

A non-blocking checkpointing policy was adopted to backup and restore the state of the application processes. The checkpointing mechanism forks an exact copy (thread)of the application program, this thread performs all the checkpointing routines without suspending the execution of the application code, thus significantly reducing the checkpointing overhead.

In order to co-ordinate the operation of the checkpointing mechanism in distributed computing environments, a novel approach to reliable distributed computing for messages-passing applications was devised. It takes advantage of the low failure-free overhead of coordinated checkpointing methods with logging messages that cross the recovery line to avoid blocking the application process during the checkpointing protocol. The low failure-free overhead is at the expense of a longer rollback time, which is admissible because of the extended execution time of the targeted application.

The noteworthy point of the presented approach is that a process receiving a message does not need to worry whether the received message may become an orphan or not. It is the responsibility of the sender of the message to make it non-orphan. Because of this, each process is able to perform its responsibility independently and simultaneously with others just by testing its local Boolean flag. This makes the algorithm a single phase one and thereby, in effect, makes the algorithm fast, simple and efficient.

A microreboot is the selective crash-restart of only those parts of a system that trigger the observed failure. Microreboots are largely as effective as full reboots but 30 times faster[8].

REFERENCES

[1] Wang, Y.-M.: *'Consistent global checkpoints that contain a given set of local checkpoints'*, IEEE Trans. Comput., 1997, 46, (4), pp. 456–468

[2] Koo, R., and Toueg, S.: *'Check pointing and rollback-recovery for distributed systems'*, IEEE Trans. Software Eng., 1987, 13, (1), pp. 23–31

[3] Venkatesan, S., Juang, T.T.-Y., and Alagar, S.: *'Optimistic crash recovery without changing application messages'*, IEEE Trans. Parallel Distrib. Syst., 1997, 8, (3), pp. 263–271

[4] Cao, G., and Singhal, M.: *'On coordinated check pointing in distributed systems'*, IEEE Trans. Parallel Distrib. Syst., 1998, 9, (12), pp. 1213–1225

[5] Pradhan, D.K., and Vaidya, N.H.: *'Roll-forward check pointing scheme: a novel fault-tolerant architecture'*, IEEE Trans. Comput., 1994, 43, (10), pp. 1163–1174

[6] Gass, R.C., and Gupta, B.: *'An efficient check pointing scheme for mobile computing systems'*. Proc. ISCA 13th Int. Conf. Computer Applications in Industry and Engineering, Honolulu, USA, November 2000, pp. 323–328

[7] Gupta, B., Banerjee, S.K., and Liu, B.: *'Design of new roll-forward recovery approach for distributed systems'*, IEE Proc., Comput. Digit. Tech., 2002, 149, (3), pp. 105–112

[8] George Candea, Aaron B. Brown, Armando Fox and David Patterson: *'Recovery-Oriented Computing: Building Multitier Dependability'*,

[9] IEEE Computer Society, November 2004, pp 60-67

[10] B. Gupta, S. Rahimi and Z. Liu : *'Novel low-overhead roll-forward recovery scheme for distributed systems'*, IET Comput. Digit. Tech., 2007, 1, (4), pp. 397–404

[11] T. Osman and A. Bargiela: *'FADI: A fault tolerant environment for open distributed computing'* , IEE Proc.-Softw., Vol. 147, No. 3, pp 91-99