International Journal of Research in Advanced Engineering and Technology
ISSN: 2455-0876; Impact Factor: RJIF 5.44
www.newengineeringjournal.in
Volume 3; Issue 2; May 2017; Page No. 15-17

Real-Time fault tolerance mechanism for distributed systems

Ozor Godwin O, Nwobodo Lois O, Ozioko Erasmus I

Computer engineering, Enugu State University of Science and Technology, Enugu, Nigeria

Abstract
Network of computers or other systems step from simple to complex. When a system is referred as complex and stochastic then the challenges of availability, dependability, stability and reliability become serious indicators for effective performance. Fault-tolerance plays a crucial role towards achieving dependability, reliability, stability and the fundamental requirement for the design and development of effective and efficient fault-tolerance mechanisms. It is also important that the power, weight, space and cost constraints of systems are addressed by efficiently using the available resources for fault-tolerance. In life critical mission systems, reliability is a great option, hence this paper investigate a fault tolerance mechanism using checkpointing in distributed systems.

Keywords: distributed checkpointing, fault-tolerance, availability, reliability

Introduction

The need for safety and longevity is paramount in system design and operation. Capacity and working principles should be considered to be effective or high performance measure or indicator for a viable system. Well organized computer cluster was investigated in this paper as a better measure to minimize downtime or failure thereby increasing reliability and availability of a system. A computer cluster consists of a set of loosely or tightly connected computer that work together so that, in many respects, they can be viewed as a single system. Unlike grid computers, computer clusters have each node set to perform the same task, controlled and scheduled by software. To enhance reliability, checkpointing coupled with fault detection scheme was adopted for fault tolerance design. Checkpointing is a technique to add fault tolerance into computing systems. It basically consists of saving a snapshot of the application's state, so that it can restart from that point in case of failure. This is particularly important for long running applications that are executed in failure-prone computing systems. There are two main approaches for checkpointing in systems: coordinated checkpointing and uncoordinated checkpointing. In the coordinated checkpointing approach, processes must ensure that their checkpoints are consistent. This is usually achieved by some kind of two-phase commit protocol algorithm. In uncoordinated checkpointing, each process checkpoints its own state independently. It must be stressed that simply forcing processes to checkpoint their state at fixed time intervals is not sufficient to ensure global consistency. The need for establishing a consistent state (i.e., no missing messages or duplicated messages) may force other processes to roll back to their checkpoints, which in turn may cause other processes to roll back to even earlier checkpoints, which in the most extreme case may mean that the only consistent state found is the initial state (the so-called chain reaction known as domino effect) may occur. In high performance computing, systems are built from highly reliable components. However, the overall failure rate of systems increases with component count. Nowadays, computer systems have a mean time between failures (MTBF) measured in hours or days (Reed D, 2004) [8] and fault tolerance (FT) is a well-known issue. Long running large applications rely on fault-tolerant (FT) techniques to successfully finish their long executions. Checkpointing is a popular technique in which the applications save their state in stable storage, frequently a parallel file system (PFS); upon a failure, the application restarts from the last saved checkpoint. Checkpointing is a relatively inexpensive technique in comparison with the process-replication scheme that imposes over 85% of overhead.

Related Work

Checkpointing with rollback-recovery and reserved rugged dynamic storage space is a well-known technique for fault-tolerance in distributed computing systems. There has been much work in the field of distributed checkpointing and rollback recovery. There are sound checkpointing protocols survey by (Elnozahy M *et al*, 1996) [1]. Though it lacks scalability properties for future and expanded supercomputers. With respect to deep research investigated and conducted, we observed that the largest known checkpointing system demonstrated before now is CLIP (Chen Y *et al*, 1997) [5], which ran experiments on a 128 processors Intel paragon machine with 32MB of memory on each node. Many other implementations are available at various levels of abstraction: application-level (Beguelin A *et al*, 1997) [2], compiler supported (Bronevetsky G, 2003) [3] runtime-library supported by (Agbaria A and Plank J. S, 2000) [4], operating system supported (Sankaran S *et al*, 2003) [6] and hardware-assisted (Prvulovic M *et al*, 2002) [7]. Clearly, each of these abstractions have their respective advantages and disadvantages, and one has to make trade-offs with respect to factors like user-transparency, portability, flexibility and granularity of the checkpointing desired. An excellent description of complete design space and associated trade-offs is given in a recent study by Sancho *et al*, 2004. In our work however, the focus is on the real-time, performance and scalability of the checkpointing technique, which we believe that the cooperative checkpointing approach holds most promise.

Most traditional distributed multiprocessor recovery schemes are designed to tolerate an arbitrary number of failures. Hence they store their checkpoint data in a central stable storage. The central stable storage usually has its own fault tolerance techniques to protect it from failures. But the bandwidth between the processors and the central stable storage is usually very low. Several experimental studies presented in (Silva M and Silva G) have shown that the main performance overhead of checkpointing is the time spent writing the checkpoint data to the central stable storage.

Scalability of the Fault Tolerance

Today's parallel architectures are usually robust enough to survive node failures without suffering complete system failure. However, most of today's high performance computing applications cannot survive node failures and, therefore, whenever there is a node failure, have to abort themselves and restart from the beginning or a stable-storage-based checkpoint.

In this paper, we explore how to build fault tolerant high performance computing applications using checkpointing so that these applications can tolerate partial process failures with lower overhead and better scalability than using the traditional checkpoint/restart approach. We analyze existing checkpointing techniques and introduce several new encoding strategies into checkpointing to improve the scalability of the techniques. Experimental results demonstrate that our fault tolerance approach can survive a small number of simultaneous processor failures with low performance overhead.

Assume checkpointing is performed in a parallel system with p processors and the size of checkpoint on each processor is m bytes. It takes $\alpha+\beta x$ to transfer a message of size x bytes between two processors regardless of which two processors are involved and α is often called latency of the network. $\frac{1}{\beta}$ is called the bandwidth of the network.

Assume the rate to calculate the sum of two arrays is γ seconds per byte. We also assume that it takes $\alpha+\beta x$ to write x bytes of data into the stable storage. Our default network model is the duplex model where a processor is able to concurrently send a message to one partner and receive a message from a possibly different partner. The more restrictive simplex model permits only one communication direction per processor.

We also assume that disjoint pairs of processors can communicate each other without interference each other. By simply organizing all processors as a binary tree and sending local checkpoints along the tree to the checkpoint processor, the time to perform one checkpoint, T_{binary}, can be represented as

$$T_{binary} = 2\log p.m(\beta+\gamma) + 2\log p.\alpha \qquad 1.1$$

Note that, in a typical checkpoint/restart approach, the time to perform one checkpoint, $T_{checkpoint/restart}$, is

$$T_{checkpointing/restart} = p.m\beta + p.\alpha \qquad 1.2$$

Therefore, by eliminating stable storage from checkpointing and replacing it with memory and processor redundancy, cooperative checkpointing improves the scalability of checkpointing greatly on parallel and distributed systems.

Evaluation analysis

We run a set of simulations to evaluate the performance of our implementation of distributed checkpointing. We compare that performance to conventional checkpointing. In conventional checkpointing one task saves its data segment, and then all tasks cooperate to save the distributed array. Conversely, in distributed checkpointing, each task saves its entire data segment. The reverse operations occur when restarting an application from a checkpointed state. We consider the following performance parameters:

Size of Checkpointing Code

This is the size of the program, seen as embedded in the large operating program. Large checkpointing code will led to heavy delay operation and so much interference.

Size of saved state

This is the total size of all files necessary to capture the state of a distributed application. For distributed checkpoint this consists of one file with the data segment of one task, plus one file for each distributed array.

Checkpoint time

This is the time to write the state of the application to the file system.

We use a blocking checkpoint: the application does not continue execution until its state has been written to the file system. For distributed applications, the selected task first writes its data segment. After that, each distributed array is written in sequence. Note that distributed array I/O operations in distributed (reads and writes) include both file I/O operations as well as data redistribution.

Restart time

This is the time to restart the execution of an application from a saved state. For distributed applications this includes the time for each task to load its data segment from the single saved segment, plus the time for the application to read the distributed arrays.

We started from hand-optimized versions of these benchmarks designed to run on the SP using MPL message-passing. We then added distributed constructs to make them reconfigurable and checkpointable. We also made the original applications checkpointable using conventional checkpointing. The simulation shows that the operation can run on both 8 and 16 processors and a checkpoint was taken at mid-point of execution. Restarts were performed from the state saved at mid-point. The table 1 shows the rate of the system operation.

Table 1: Checkpoint and Restart operation per (16 processors)

Checkpoint				Restart			
Total		Data Segmen	Arrays	Total		Data Segmen	Arrays
Time (s)	Rate (MB/s)	Rate (MB/s)	Rate (MB/s)	Time (s)	Rate (MB/s)	Rate (MB/s)	Rate (MB/s)
20.5	8.8	10.5	6.8	50.5	35.3	60.2	8.5
22.7	9.1	10.8	7.1	48.8	37.1	64.4	7.9
25.1	7.6	8.9	6.9	47.9	34.8	70.1	8.6
23.0	6.9	9.0	6.6	50.2	36.0	67.5	8.3
26.2	8.3	7.6	8.1	51.0	38.7	59.0	7.5
23.6	7.5	8.0	7.3	51.8	35.7	68.8	8.1
23.4	8.2	8.2	7.6	49.8	30.1	61.7	8.3

Discussions on Checkpointing

We have compared the performance of checkpoint/restart of distributed applications to that of a conventional applications. Although the checkpointing mechanism used for the conventional version is a rather straightforward and somewhat naive (each task takes a separate checkpoint), it is similar to the approach described in literature by others. We note that, in our simulation results we have not considered some of the optimizations that the state-of-the-art in distributed checkpointing may provide. For example, optimizations can be applied to reduce the amount of data saved. These optimizations range from application-independent optimizations (data compression, incremental checkpointing that saves only modified pages) to compiler-based optimizations that can tailor checkpointing to a specific application (detection of killed variables, computation of array sections accessed). While these optimizations can be equally applied to distributed checkpointing, they can erase much of the difference in saved state size observed. We claim, however, that the distributed and its programming model can still bring some reduction of saved state size to an important class of applications.

Conclusion

Disturbed checkpointing compares favorably to a straightforward implementation of checkpointing for applications where the run-time system has no knowledge of the distributed data structures as in fig 1and 2. We have shown the advantages of distributed checkpointing in terms of size of saved state, time to checkpoint, and time to restart the application.

Checkpointed applications can be restarted on a smaller system while failed processors are being serviced, thus reducing application down-time. Applications can also be started on a larger system, to take advantage of more processors. This shows a significant positive impact on the performance and usability of large parallel systems in the presence of component failures.

The key point was that the system with checkpointing fault tolerance mechanism has high availability compared to the system without fault tolerance.

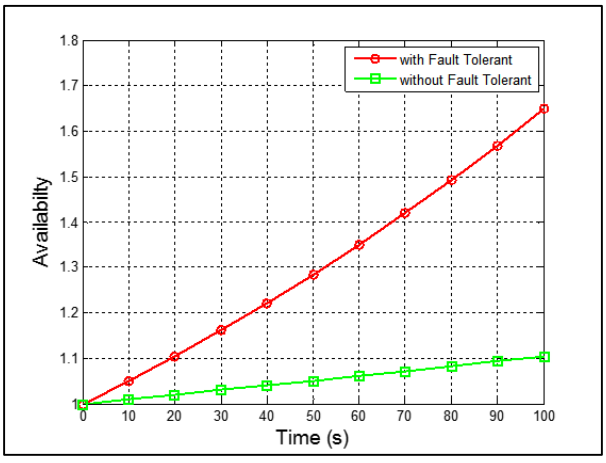

Fig 1: Availability assessment based on fault tolerance

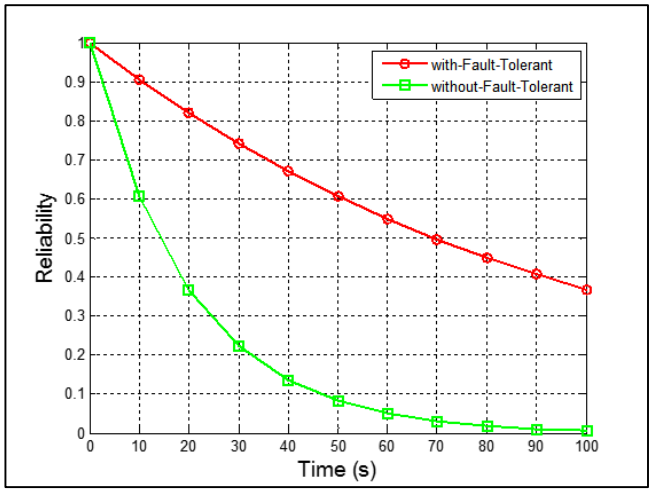

Fig 2: Reliability assessment based on fault tolerance

Reference

1. Elnozahy M, Alvisi L, Wang Y, Johnson D. A survey of rollback-recovery protocols in message passing systems, Tech. Rep. CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1996.
2. Beguelin A, Seligman E, Stephan P. Application level fault tolerance in heterogeneus networks of workstations. Journal of Parallel and Distributed Computing. 1997; 43: 147-155.
3. Bronevetsky G, Marques D, Pingali K, Stodghill P. Automated application-level checkpointing of mpi programs, in In Principles and Practice of Parallel Programming, 2003.
4. Agbaria A, Plank JS. Design, implementation, and performance of checkpointing in netsolve, in In Proceedings of the International Conference on Dependable Systems and Networks, 2000.
5. Chen Y, Plank JS, Li K. CLIP: A checkpointing tool for message-passing parallel programs, in SC97: In Proceedings of the Supercomputing, 1997.
6. Sankaran S, Squyres J, Barrett B, Lumsdaine A, Duell J, Hargrove P, Roman E. The design and implementation of Berkeley lab's Linux checkpoint/restart, in Los Alamos Computer Science Institute (LACSI) Symposium, 2003.
7. Prvulovic M, Zhang Z, Torrellas J, Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. in In Proceedings of the International Symposium on Computer Architecture, 2002.
8. Reed D. High-End Computing: The Challenge of Scale, Director's Colloquium, LANL, 2004.