International Journal of Engineering Science Invention
ISSN (Online): 2319 – 6734, ISSN (Print): 2319 – 6726
www.ijesi.org Volume 2 Issue 5 ‖ May. 2013 ‖ PP.64-69

Adaptation Mechanism for Managing Grid Resources

[1.] Dr D. R. Aremu [2.]Faki A.Silas

[1,2]University of Ilorin, Ilorin, Faculty of Communication and Information Sciences,
Department of Computer Science.

ABSTRACT: *As Grid architecture provides resources that fluctuates, application that should be run in this environment must be able to take into account the changes that may occur. This application must adapt to the changes in Grid environment. Checkpointing is one way to make applications responds to these changes. Though, this can not be done without incurring checkpoint overheads. To reduce these checkpoint overheads and make application run optimally, checkpoint interval and total time to release an executing job in resources must be taken in to consideration. This can be efficiently achieved by use of exception handling. Exception handling, though has its roots in programming language design, can be used to handle fault in a grid environment. The combination of exception handling model with checkpoint parameters can perform optimally in reduction faults in grid resources.*

Keyword: *Checkpoint, Exception Handler, Grid, Commuting Resources, Fault Tolerance.*

I. INTRODUCTION

Grid computing arose in the 1990's, in the supercomputing community with the goal of making underutilized computing resources easily available for complex computation across geography distributed sites. The main goal of grid is not to buy new resources but to borrow the power of computational resources from where they are not in use. Management of grid resources is increasingly becoming very complex due to resources that are geographically located, heterogeneous in nature, own by different individuals or organizations with their different policies, access and cost models, have dynamically varying loads and availability. As a result, the chances of failure is on the increase. As a matter of fact, in the grid systems, resource failure is the rule rather than exception. Resources could be turned off or simply given to another user for other reasons. Also bandwidth or service time of some computing resources could be extremely low. In the midst of all these challenges, grid applications are expected to accomplish their task as if there are no complexities. To counter the problem of volatility in the grid environment, system designers have come up with various methods such as Checkpoint/Restart, Replication, Retry, Message logging etc. Checkpoint/Restart is an interesting fault tolerance management method wherein the snapshot of the program execution is taken at regular intervals and saved in a stable storage. When a node failure happens, the stored snapshot (or process image) is used to restore the failed process either on the same node or elsewhere. Replication is another interesting fault tolerance mechanism which relies on multiple replicas of the same process running on multiple nodes. The idea of fault tolerance is simple: if one of the nodes fails, the replica node(s) will continue the execution and thus make the system robust up to some level. With the advantages of fault tolerance management, comes a disadvantage in the form of overhead. The fault tolerance overhead is the amount of time an application spends in enforcing fault tolerance policies. Every time a checkpoint-enabled application takes a snapshot of its process image, it incurs an overhead that is unavoidable. So, when implementing fault tolerance, the inevitability of incurring the overhead has to be kept in mind. To minimize the amount of overhead in checkpointing, this paper presented a checkpoint interval model that is simple and has minimal overhead and a recovery process that is fast and efficient using exception handlings. The rest part of the paper is organized as follows: Section 2 discussed the related work, while section 3 presented fault tolerance management designs using Checkpoint Interval model. In section 4 the simulation results of the checkpoint interval model was discussed, while sections 5 concluded the paper, and section 6 presented future work.

II. RELATED WORK

Interest in adaptive computing systems has increased dramatically in the recent past few years, and a variety of techniques now allows software to adapt dynamically to its environment. According to Jaqualine (2006), dynamic adaptation and reconfiguration of software date back as earliest days of computing, when self-modifying code supported runtime program optimization and explicit management of physical memory.
According to Jaqualine (2006), there are three general approaches to implement modeling and designing mechanism for dynamic adaptation. These are: **Parameter adaptation**. Parameter adaptation modifies programmed variables that determine behavior. The internet Transmission Control Protocols (TCP) is an often-

cited example. TCP adjust its behavior by changing values that control window management and retransmission in response to apparent network congestion. The weakness of this approach is that it does not allow new algorithms and components to be added to an application after the original design and construction has been done. It can tune parameters or direct an application to use a different existing strategy but it cannot adapt new strategies. However, it offers the advantage that the adaptation can be performed with good performance.
Code (agent or component) migration. Code migration can be defined as the capability of a distributed application to relocate its component at runtime. In Lang and Mitsuru (1998), this is referred to as logical mobility of the user service as in mobile computing. In general, component migration may involve the code of a software component (e.g. the code of a class) or even some combination of code and state, often referred to as a mobile agent. ***Compositional Adaptation***. Compositional adaptation result in the exchange of algorithmic or structural part of the system, in order to improve a program's fitness to its current environment.

By contrast, compositional adaptation exchange algorithms or structural system components with others that improve a program's fit to its current environment. With compositional adaptation, an application can adapt new algorithms for addressing concerns that were unforeseen during development. This approach has a larger adaptation scope than the parameter adaptation, as it does not only enable simple code tuning programmed during design time but also cope with adaptation types unforeseen during the original design and construction.

A dynamic adaptation considers that it should occur in order to maximize the equation between the application and its execution environment (Jeremy, Francoise & Jean, 2005). This means that the purpose of dynamic adaptation is to optimize the application whenever its execution environment changes. Furthermore, application can adapt itself anywhere in the execution part. It can be either in the past state or at a state in the future.

II.1. Fault Tolerance: Fault tolerance is a survival attribute of a computer system. Paul and Jie (2003) stated that, the function of fault tolerance is to "*....To preserve the delivery of expected service despite the presence of fault caused errors within the system itself. Errors are detected and corrected, permanent fault are located and removed while the system continue to deliver accepted service* "

Fault tolerance is the ability of an application to operate in the presence of software and hardware failures, i.e. processors and network crashes. A large number of research effort has already been develop on fault tolerance. Various aspects that has been explore include design and implementation of fault detective services as well as the development of failure prediction and recovery strategy. Though, both methods seem to improve system performance in the presence of failures, their effectiveness largely depends on tuning runtime parameters such as the *checkpoint interval* and *number of replicas*. The work on grid fault tolerance can be divided into pro-active and post-active (Antony, Theresa, Sumathi & Antony, 2010). In proactive mechanism, the failure consideration for the grid is made before the scheduling of the job, and dispatched with hope that the job does not fail. Whereas post-active mechanism handles the job failure after it has occurred. Of those that look into this issue, most works are post-active in nature and deal with failure through grid monitoring (Medeiros, Cirne, Brasilairo & Sauve, 2003).

II.2. Fault Tolerance Techniques: Some of the important approaches for implementing fault tolerance in distributed applications are discussed below.

Ii.3. Check Pointing: According to Tanenbaun and Van (2002), the most popular fault-tolerance mechanism is check pointing which means the periodical saving of the state of the application on stable storage, a device that can survive failures (usually one or more hard disks). The information stored on the stable storage is called a checkpoint. After a crash, the application is restarted from the last checkpoint rather than from the beginning. Elnozahy, Alvisi, Wang and Johnson (2002) stated that, Checkpointing comes in three varieties: uncoordinated, coordinated and communication induced checkpointing.
The main advantage of checkpointing is that it is very general technique which can be applied to any type of parallel applications. Though it has disadvantages that it causes execution time overheads, even when there are no crashes (This can be reduced by incremental checkpointing). The overhead is dependent on the frequency at which checkpoints are taken and this depends on the programmer.

II.4. Parameter affecting the Performance of Checkpoint System.

a. The various factors that affect the performance of checkpoint are as follows,Checkpoint Interval: Determine the efficiency of the checkpoint system.
b. The checkpoint interval has to be optimal to achieve high performance.
c. Time to checkpoint: determine the bulk of the overhead incurred in checkpointing. We get better performance as the checkpointing overheads reduces.
d. Time to restart: determine the overhead incurred in restarting an application process on Another node when failure happens. This too is responsible for degradation of performance as its value increase.

II.5. Message Logging

An alternative fault-tolerance technique is message logging. During failure free operations, each process logs sent or received messages (depending on the variant of messages logging algorithms) from other processes (Elnozahy et al, 2002). After a failure, the crashed process is re-executed and the logged messages are replayed. Message logging is typically combined with checkpointing to reduce the amount of re-execution needed. Message logging enables the system to recover beyond last checkpoint. Also, message logging is used to provide the application ability to interact with the outside world. Though, it is used less often than checkpointing.

According to Bouteiller, Lemarinier, Krawezik and Capello (2003), Message logging schemes comes in three flavors: pessimistic message logging, optimistic message logging and casual message logging. Message logging is very general technique but it can cause high execution time overhead. It also affects communication throughput and latency.

II.6. Replication

In replication, multiple copies of the same task/process are run on separate processors. If one of the copies crashes, other copies are used. This technique can be used not only for tolerating crashes failure but also Byzantine failures. The technique is suitable for system of which high availability is required since the recovery is fast. Replication is often used in hardware-based fault tolerance.

II.7 Retry: Another technique used for preventing fault tolerance is retry: re-computing parts of the work that were lost in a crash. This technique is good for applications that adhere to functional programming paradigm. A functional programming application consists of functions with no side effects. There is no notion of a global state and a result of a function depends solely on its input parameter. Example of such that uses functional programming is master-worker application.

II.8 Exception Handling: According to Brian (2006), exception handling has its roots in Programming Language Design but can be viewed in more general terms. It is of course, at best just another divide and conquers method for coping with complexities. It is a well design language construct that enables Programmers to simplify their task by identifying and dealing separately with various predictable but uncommon situations during complexities and errors. Each such situation, perhaps is describe by a set of logical preconditions, can have it own separate exception handler associated with it, design to deal with just this precondition, and if possible to achieve a post condition in focus.

In summary, exception handling technique though by no means is a panacea, can be a powerful aid to structuring and hence simplifying very complex situations and the design of systems that have to cope with faulty situations.

III. FAULT TOLERANCE MANAGEMENT DESIGNS

This section presented three designs for fault tolerance management. These designs are: checkpoint interval design, exception handler algorithm, and fault index update algorithm.

Exception Handler As William Shakespeare say and I quote “if they’re running and they don’t look where they’re going I have to come out from somewhere and catch them”. It is necessary to catch them before they go out of hand and this is the function of exception handler. If an exception (fault) occurs as a result of a statement in a try block which may be due to, computing resource being switch off, low bandwidth, service time of resource being reduces due to hardware problem etc, the try block terminates immediately. Next the program searchers for the first catch handler that can process the type of fault that occurred.

It the try block completes its execution successfully (i.e. no fault in job execution), the program ignores the catch handlers and program control continues after the first statement after the catch block.

Exception handler Algorithm

N: number of checkpoint to be made in a job
n: number of checkpoint already made
While (N! = 0)
{try { 1. is job computing in the resource} catch { 1a. resent job to the scheduler at the nth checkpoint
1b. notify the user that his job has been rescheduled to resource i}N -= 1}

III.1 Fault Index Manager: Fault index manager maintains the fault index value of each resource which indicates the failure rate of the resources. The fault index of a grid resource is increment every time the resource does not complete the assigned job within the deadline and also on resource failure. The fault index manager updates the fault index of a grid resources using index update algorithm.

III.2 Fault Index Update Algorithm

a. **IF** checkpoint manager receives the job completion result from resources THEN
1. **IF** resources fault index >= 1 **THEN**
2. Send a massage to fault index manager to decrement the fault index of resource that completes the assigned job.
3. Send details of the finished job to the scheduler**END IF**
b. **GOTO** step 3
c. **END IF**
d. **IF** checkpoint manager receives the job failure message from resources **THEN**
e. Send message to fault index manager to increment the fault index of resources that has fails to complete the assigned job.
f. Send a message to checkpoint server, whether there is any checkpoint result of this job.
g. **IF** checkpoint result of the job exist in the checkpoint server **THEN** Submit the remaining part of job after last checkpoint received to the scheduler for rescheduling **GOTO** step 3 **END IF**
h. **IF** checkpoint result of the job does not exist in the checkpoint server **THEN** Submit the job from start to the scheduler for rescheduling. **GOTO** step 3 **END IF END IF**

III.3 EXIT: Checkpoint Interval Designed Model

Checkpoint Setter Algorithm

Input: Fault Index of a resource
F: Fault Index of the selected grid resources
F(i), i= 0,2, . . . , N, are integers such that F(0) < F(1) . . . < F(N)

1. IF (F == F(1)) **THEN**
(i) The job is queue to that resources with a checkpoint interval 1
(ii) GOTO step 6.

2. IF (F ==F(2)) **THEN**
(i) The job is queue to that resources with a checkpoint interval 2
(ii) GOTO step 6.

3. IF (F == F(3)) **THEN**
(i) The job is queue to that resources with a checkpoint interval 3
(ii) GOTO step 6.

4. IF (F == F(N)) **THEN**
(i) Then the job is queue to that resources with a checkpoint interval N
(ii) GOTO step 6.

5. IF (F > F(N)) **THEN**
(i) Remove resources from available resources and label it as unavailable resources i.e. no job is assigned to that resources
(ii) Add the job to the unassigned job list and reschedule it.

6. EXIT

3.1 Experimental Design For Evaluating Checkpoint Interval

The duration of time interval checkpoints are to be set as the job progress depends on the user.
The models below determine the number of checkpoint to be set base on the fault index of a resource.

$$J_{endT} = J_{startT} + (n - 1)\, C_i \quad . \quad . \quad . \quad \textbf{Model I}$$

$$J_{endT} = J_{startT} * C_i^{(n - 1)} \quad . \quad . \quad . \quad \textbf{Model II}$$

Where J_{endT} is the end time of execution of job j,
J_{startT} is the start time of execution of job j,
n is the number of checkpoint of job j with the execution time,
C_i is the checkpoint interval at which checkpoint is to be set.

Checkpoints are determining based on the fault index of a resource, the higher the fault index, the closer checkpoint interval is set on the resources.

The models set checkpoint in an arithmetic or geometric manner. Though in data intensive applications, the checkpoint file might become huge. If the amount of space on stable storage is limited, it is necessary to prevent the checkpoint file from growing too much by implementing checkpoint file compression.

IV. SIMULATION RESULT

There are three basic methods that can be use to get information on real world Systems: Experimentation, Analysis and Simulation.

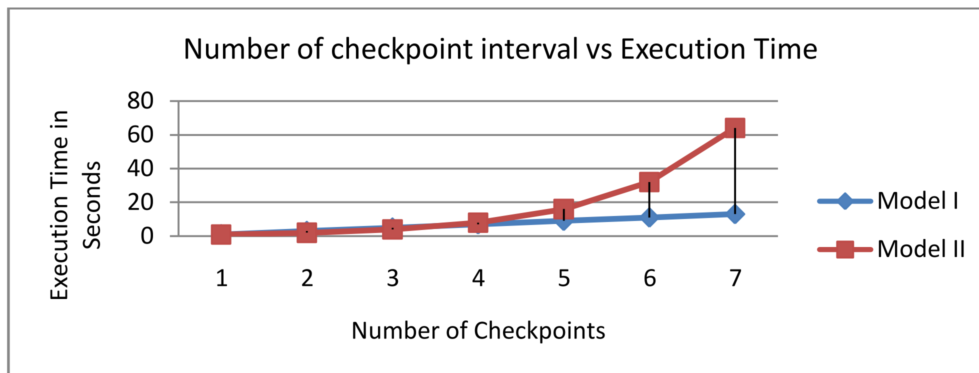
Experiment seem to be more accurate but sometimes it is costly, time consuming and sometimes dangerous. Analysis is typically base on heavy assumptions (mostly mathematically) that are rarely true in practical real life situations. Simulation sometimes is the only way out because it can represent a real world with few assumptions, less cost and easily modifiable.

The presented strategy is evaluated using simulation.

Experiment 1 Determination of effects of checkpoint interval using our model.

A job is considered with start and end time. Checkpoint taking at fixed interval.

The graph below is drawn using model I and II with the job start time at one minute, checkpoint interval of two seconds; seven checkpoints were considered before the job elapse. This implies that the job takes 13 seconds to complete using model I and 64 seconds to complete using model II.



Graph 1. Graph of checkpoint interval vs. time of execution.

Graph 1 shows the performance of checkpoint interval as execution time progresses using the presented models. It can be observes that both models exhibit close related behavior up to the fourth checkpoint interval, the execution time and the number of checkpoint made is almost the same. After which model II made a sharp increase in time of execution. Though, the numbers of checkpoints made are the same, the time used in execution of the job is not the same. Model II execution time is more than model I. This implies that Model I will make more checkpoint than model II thereby incurring more checkpoint overhead. In terms of recovery, model II will do more recomputing of failed job because the checkpoints are far apart in the terms of time. Conclusively, model I is more suitable for computing resources that have high fault index because of its checkpoint interval are close.

V. CONCLUSION

Fault tolerance using checkpointing is very popular but has its own price tag, checkpoint overhead. To minimize the checkpoint overhead, we carefully evaluated the checkpoint interval using two models (model I and Model II).

Model I as stated in earlier is arithmetic in nature. The checkpoint interval increase at arithmetic sequence in relation with time of job execution of the computing resource. The checkpoint interval here is small in nature which means the system will spend some useful time checkpointing and thus incur a high checkpoint overhead. This model is applicable to resources with high fault index from the fault index manager.

Model II checkpoint interval increase geometrical with time of execution of the computing resource. This model make little checkpoint intervals over time thereby recording a little checkpoint overhead. Though it has a disadvantage of executing a large portion of code over and over again when failure occurs. This model is most suitable for a resource with low fault index.

Conclusively, we said the fault index determine the model to be implemented.

VI. FUTURE WORK

This work is done purely with simulation. In the future, it is planned to explore the potentials of these algorithms by embedding them in a real world grid computing environments.

REFERENCES

[1]. Antony, L. Theresa, A. Sumathi, G & Antony, D. S. (2010), Dynamic Adaptation of Checkpoints and Rescheduling in Grid Computing, International journal of Computer Application (0975-8887) Vol. 2-No. 3.
[2]. Bouteiller, A. Lemarinier, P. Krawezik, K & Capello, F. (2003),"Coordinated checkpoint versus message log for fault tolerant MPI,"Proceedings of IEEE International Conference on Cluster Computing, pp.242 250.
[3]. Brain, R. (2006), University of Newcastle upon Tyne, June.
[4]. Elnozahy E. A. Alvisi, L. Wang, Y.M. & Johnson, D.B. (2002), A survey of Roll-Back recovery Protocols in massage -passing System, ACM Computing Survey, September.
[5]. Foster, I. (2002) What is Grid? A Three Point Checklist, Argonne National Laboratory and University of Chicago, 20 July.
[6]. Jaqualine, F. (2006), Mobility and Adaptation Enabling Middleware (Madam) SINTEF, D2.2, December.
[7]. Jeremy, B. Francoise, A. & Jean, L. P. (2005), Dynamic Adaptation for Grid Computing, IRISA/INSA de Rennes, Rennes France, IRISA/Universite de France.
[8]. Lang,D.B & Mitsuru, O. (1998), Programming and deploying mobile Java agent with aglet, Addison Wesley Longman Publishing Company ISBN 0201325629.
[9]. Medeiros, R. Cirne, F. Brasilairo, F & Sauve, J. (2003), Fault in grid: Why are they so Bad and What can Be Done about It. In proceeding of the Fourth International Workshop on grid Computing, (Grid'03).
[10]. Paul, T. Jie, X. (2003), Fault Tolerance within a Grid Environment, University of Durham DHI 3LE, United Kingdom
[11]. Tanenbaun, A.S. & Van Steen, M. (2002), Distributed Systems, Principles and Paradigms, Prentice Hall.