International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395 -0056
Volume: 03 Issue: 05 | May-2016 www.irjet.net p-ISSN: 2395-0072

LAYERED FAULT TOLERANCE FOR COMPUTE CLUSTERS

A. PADMA PRIYA[1], G. RAJESH KRISHNA[2]

[12]Assistant Professor, ECE dept., Nalla Malla Reddy Engineering college, Hyderabad, INDIA.

Abstract *- Clusters of message-passing computing nodes provide high-performance platforms for distributed applications. Cost-effective implementations of such systems are based on commercial off-the-shelf (COTS) hardware and software components. We present a layered approach to providing fault tolerance for message-passing applications on compute clusters that are based on COTS hardware components, COTS operating systems, and a COTS application programming interface (API) for application programmers. This approach relies on highly-resilient cluster management middleware (CMM) that ensures the survival of key system services despite the failure of cluster components.*

Key Words: *Clusters, Commercial off-the-shelf, cluster management middleware, Fault tolerance.*

1.INTRODUCTION

Fault tolerance for high-performance distributed applications is increasingly important due to unreliable cluster nodes. As the number of nodes in a cluster increases, the probability of a single node failure at a given time also increases. Furthermore, the reliability of integrated circuits may decrease due to shrinking feature size and lower voltages. Clusters can also be deployed in harsh environments, where radiation and other conditions can cause malfunction in the hardware. Although much work has focused on making high-performance applications fault-tolerant, most of the work is concerned only with fail-stop faults, in which faulty processes crash. Such faults are easy to detect, and, given that a copy of fault-free application state exists, easy to recover from. The goal of this work is to enable distributed applications running on clusters to detect and recover from process crashes, process hangs and arbitrary faulty behavior, such as the generation of incorrect results as shown in figure 1. All of these errors are easily detected by replicating applications and comparing the outputs of the replicas. However, replication imposes a great cost—at least *n* times the resources required to maintain *n* replicas. There exist many fault tolerance mechanisms that enable detection and recovery from hangs and incorrect results while using fewer resources than replication. Our work focuses on system support needed to implement such fault tolerance mechanisms for distributed applications. This chapter describes in general terms what is needed to implement fault tolerance mechanisms for distributed applications and gives justification for implementing certain functionality as cluster management middleware (CMM) services. This paper firstly explains a general principle for the greater efficiency of application-specific fault tolerance mechanisms over fault tolerance mechanisms that work for all applications and also describes exceptions to the principle, leading to an implementation that involves multiple software layers. A way that is not application specific.

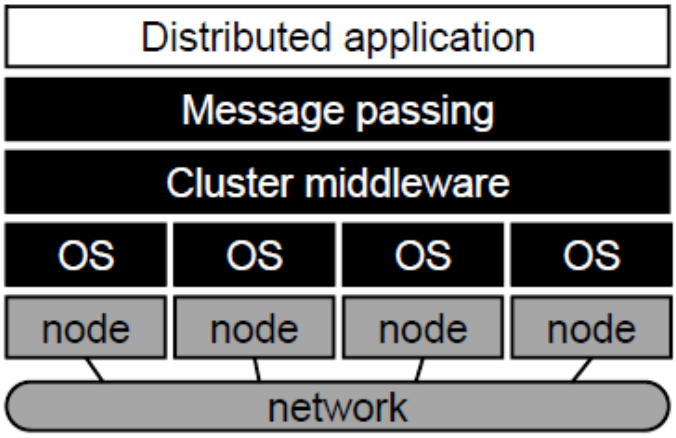

Fig -1: Typical layered architecture of cluster systems

2. END-TO-END ARGUMENTS FOR APPLICATION-SPECIFIC FAULT TOLERANCE

The general solution for implementing application fault tolerance is to introduce redundancy for the application's state. For example, check pointing is a fault tolerance mechanism that stores copies of application state made at specific points in time in an application's execution. Process-level replication is another example of a mechanism that maintains copies of the application, where replicas of applications are simultaneously executing. Both of these mechanisms can be implemented in an application-transparent way so that they work for all applications in general, regardless of the structure of the application's data or the behavior of its algorithms. The cost of fault tolerance mechanisms in terms of resource usage and performance overhead is due mainly to the maintenance of redundant state. The amount of state to maintain can be reduced significantly by taking advantage of application-specific characteristics. For example, a checkpointing mechanism may omit committing application state that can be quickly recomputed from the checkpointed state after rollback recovery; a replication mechanism may use alternative algorithms that approximate the original algorithm while using fewer computational resources. The gain in efficiency from using application specific characteristics for adding redundancy comes at the cost of implementing the mechanisms for each application or group of applications with similar characteristics. Assuming that the gain in efficiency outweighs the additional cost in implementation, we can argue that all fault tolerance mechanisms should be implemented in application-specific manner as much as possible. Implementing application-specific fault tolerance mechanisms is an instance of the end to-end argument in system design [Salt84]. The argument for end-to-end design states that certain functionality can be implemented correctly and completely only with the knowledge of the application running at the endpoints of a communication system; providing the functionality as a feature of the communication system is not possible or may be redundant. The application of the end-to-end argument to fault tolerance mechanisms involves what layers the mechanisms should be implemented at and whether the mechanisms should be specific to the applications they are supporting. Some mechanisms such as detection of node failure operate without any dependence on application-specific knowledge and would offer no benefit from application-specific knowledge. Other mechanisms benefit from application specific knowledge, as described above. Given that it is sometimes desirable to implement application-specific fault tolerance mechanisms, we must consider what layers should be involved in the implementation. In typical clusters [Agba99] employing COTS hardware and COTS OS, there are few facilities for the hardware or OS to easily determine application-specific characteristics for implementing more efficient fault tolerance mechanisms. To implement such facilities could complicate hardware and OS for the benefit of a few kinds of applications while incurring significant cost for all applications. Part of this cost may involve instrumenting OS code or adding measurement capability in hardware to measure and analyze events at run-time that indicate the application's behavior. We assume that modification of the hardware and OS to make such characteristics visible within these layers is expensive to the point that it negates the low-csto motivation of using commercial off-the-shelf (COTS) hardware and COTS OS in the first place. The remaining option is to implement application-specific fault tolerance mechanisms in user-level processes, involving CMM, application libraries, and the application code. A typical CMM manages execution of applications on a general-purpose high performance cluster

using very little knowledge of application-specific characteristics. Application-specific procedures are implemented in application libraries and application code in order to keep the CMM implementation simple. However, there are some cases where involving the CMM in implementation of application-specific fault tolerance mechanisms yields benefits. The next section describes these cases.

2.1. Involving the CMM in Implementation of Fault Tolerance Mechanisms for Applications

There are two cases where involvement of the CMM in implementing application specific fault tolerance mechanisms can be beneficial, even though there are no application specific procedures implemented in the CMM. The first case are the functions that application-specific fault tolerance mechanisms need to perform but which must be implemented in the CMM and OS. For example, an application fault tolerance mechanism may need to terminate a faulty process and spawn a new process to replace it. The CMM manages processes on behalf of distributed applications; permitting a potentially faulty application to perform these tasks could impact the health of other applications and of the cluster itself. The second case are the functions that can be implemented more efficiently in lower layers. For example, an OS immediately detects a process crash whereas a distributed application must use heartbeat messages and timeout events to detect a process that has crashed. It is conceivable that some of the functionality of the second case presented above could be implemented in an application library instead of involving the CMM. The benefit of using the CMM is that the CMM has very high reliability requirements. As the manager of the cluster and all applications running on the cluster, the CMM is a critical component of the cluster and must be reliable. When the CMM fails, the entire cluster has failed, and there is nothing an application can do to guarantee correct execution. In other words, applications running on a cluster implicitly depend on the correct operation of the CMM. Adding CMM functionality which application-specific fault tolerance mechanisms can invoke simply makes this dependence explicit.

2.2. CMM Services for Supporting Application Fault Tolerance

In this section we propose a set of services to implement in CMM in order to support fault tolerance mechanisms for distributed applications. We consider general requirements of fault tolerance mechanisms performing four actions: error detection and notification, error diagnosis, error recovery, and reconfiguration of application processes. An error is detected when a fault tolerance mechanism concludes that the state of the system or application is incorrect. The error detection mechanism must notify the system or application so that normal execution can be stopped. In error diagnosis, fault tolerance mechanisms identify and isolate the erroneous state. In error recovery, fault tolerance mechanisms restore the state to be error-free. Finally, in reconfiguration, fault tolerance mechanisms repair or replace a faulty component so that it cannot commit more errors in the future (unless it is stricken with another fault).

2.2.1. Error Detection and Notification

Detecting hangs and arbitrary incorrect behavior of a faulty application process requires application-specific knowledge. Hence, one must use application-dependent code to implement detectors for these kinds of errors. Detecting a process hang requires knowledge of how much time application algorithms take to execute. Detecting missing output, extraneous output, and incorrect output requires

knowledge of the application's specifications regarding correct output. Hangs and arbitrary incorrect behavior can be detected by replicating the entire application and comparing the outputs of the replicas; this is a special case of application-dependent code where multiple instances of the application code itself are executed. To keep the CMM simple, we refrain from implementing functionality that depend on application-specific characteristics. Therefore, we do not implement in the CMM replication or any other service that assists in detection of hangs and arbitrary incorrect behavior. Crashes and hangs both result in a failure to make progress and can be detected by application-specific code in which application processes periodically compare the progress of the application to the passage of time. As mentioned earlier in this chapter, OS software already detects process crashes. Since the OS's detection mechanism is event-based, in contrast to the application's slower timer-based mechanism, an application would respond to 37 crashes more promptly if it handled crash notifications based on the OS's detection. Although the single-node OS is not able to notify processes running on other nodes of a crash, it can notify the CMM, which in turn can notify the remaining processes of the application. Crash notification is therefore a useful service that CMM can implement without knowledge of application-specific characteristics. An application process is prematurely terminated when the node that it is running on has failed. Since the entire node is lost, this can only be detected by the CMM or the application itself, portions of which are still running on other nodes. The CMM must detect node failures using its own mechanisms since it has to keep track of available cluster resources. Since the CMM manages cluster resources, it has the information regarding which nodes each application is using. Hence, it is very simple to add to the CMM the capability of notifying the application when one of the nodes the application is using fails. In the above discussion we identified three sources of error detection: the operating system, for process crashes; the cluster manager, for node failures; and application-specific code, for hangs and corruption of application-level state. Once any error is detected, the distributed processes must be notified quickly for coordination in diagnosis and recovery.

Notification of error detection can be accomplished through the distributed application's message-passing facility. However, there are several reasons to separate the error notifications from normal messages related to the application's distributed computation. First, notifications of error detection may originate from entities other than the application itself. For example the cluster manager may detect a failed node, or the message-passing implementation may detect a broken connection. It would be intrusive to modify existing application code so that these entities are treated as communication endpoints on the same level as the application's processes. Second, an error notification should be handled as soon as possible in order to prevent the spreading of corrupted state through message-passing. In particular, processes must be able to handle such notifications even if they arrive while the process is blocked. For example, a fault-free process expecting to receive a message from a sender may be blocked in a blocking message receive operation at the time of the error. The error may have affected the sender such that it fails to send the message that the receiver expects. The fault-free receiver in the blocking operation must be interrupted by the error notification so that it can execute diagnosis and recovery mechanisms. The above discussion leads us to a solution whereby the CMM provides a reliable asynchronous communication service for distributing error notifications. The service enable communication between the cluster manager and the application, and it enables one application

process to interrupt other application processes, even if they are blocked in message-passing operations.

2.2.2. Diagnosis

Diagnosis is the process of identifying the faulty components in a system. For a distributed application, the "components" are the individual processes. Hence, one way diagnosis can be implemented by applications is for application processes to perform system level diagnosis [Prep67], where the processes of the application test each other. Producing a correct diagnosis in a distributed system is complex because faulty components can fail to send messages they are supposed to send, send more messages than they are supposed to send, or send incorrect messages. With a fault-tolerant CMM, the cluster manager is a reliable and trusted entity that can potentially simplify the diagnosis process. For example, the reliable trusted central cluster manager can provide a very simple mechanism for identifying a majority vote among diagnoses produced by the application processes and transmitting the results to all fault-free processes.

2.2.3. Error Recovery

There are two ways to restore error-free state: rollback recovery and roll-forward recovery. In rollback error recovery, [Camp86] an earlier error-free state of the application, is restored, and computation resumes from that earlier state. The end-to-end argument applies to rollback because the application programmer has the most awareness about what application state is critical for correct rollback and at what points in the execution such state should be committed to reliable storage. In roll-forward error recovery, the application replaces its erroneous state with newly created correct state and continues execution. The new correct state may not have been reached in the past, and it may also not have been reached had there been no error. Knowing how to create a correct state involves application-specific knowledge, so the end-to-end argument applies to the implementation of roll-forward recovery.

2.2.4. Reconfiguration of Application Processes

Reconfiguration consists of actions taken to prevent a faulty component from causing another error. In terms of processes of a distributed application, reconfiguration may involve removal of processes diagnosed to be faulty, and replacement of faulty processes with fault free processes. The CMM manages processes on behalf of applications to maintain the health of the cluster and to prevent an application from interfering with the processes of another application. Therefore, implementing reconfiguration of application processes must involve communication between the application and the CMM.

2.3. A Set of CMM Services for Supporting Application Fault Tolerance

This paper explains why some functionality related to application fault tolerance can be easily implemented in the CMM, beneath the application layer, while other functionality are best left to the application-specific code for implementation. To keep CMM algorithms simple and reliable, we avoid introducing application-specific routines to the CMM. Instead, in defining a set of services to add to the CMM we consider only functions that must be implemented by the CMM and functions that are more efficiently implemented by the CMM and system software. We propose in this chapter the following set of CMM services that can be implemented to support application fault tolerance:

(1) An asynchronous communication facility enabling communication between applications and the cluster

manager and enabling interruption of application processes for error notification.

(2) Error detection for prematurely terminated processes due to crashes and node failures.

(3) A reliable voting service.

(4) A service to terminate an application process and

(5) A service to spawn an application process.

3. CONCLUSION

Fault-tolerance is achieved by applying a set of analysis and design techniques to create systems with dramatically improved dependability. As new technologies are developed and new applications arise, new fault-tolerance approaches are also needed. In the early days of fault-tolerant computing, it was possible to craft specific hardware and software solutions from the ground up, but now chips contain complex, highly-integrated functions, and hardware and software must be crafted to meet a variety of standards to be economically viable. Thus a great deal of current research focuses on implementing fault tolerance using COTS (Commercial-Off-The-Shelf) technology. Another area is the use of application-based fault-tolerance techniques to detect errors in high performance parallel processors. Fault-tolerance techniques are expected to become increasingly important in deep sub-micron VLSI devices to combat increasing noise problems and improve yield by tolerating defects that are likely to occur on very large, complex chips.

5. REFERENCES

1. [Agba99] A. M. Agbaria and R. Friedman, ‘‘Starfish: fault-tolerant dynamic MPI programs on clusters of workstations,’’ 8th International Symposium on HighPerformance Distributed Computing, pp.167-176 (1999).

2. [Salt84]- J. H. Saltzer, D. P. Reed, and D. D. Clark, ‘‘End-to-end arguments in system design,’’ ACM Transactions on Computer Systems, vol.2, no.4, pp.277-288 (November 1984).

3. [Prep67] F. P. Preparata, G. Metze, and R. T. Chien, ‘‘On the Connection Assignment Problem of Diagnosable Systems,’’ IEEE Transactions on Electronic Computers, vol.16, no.6, pp.848-854 (Decemeber 1967).

4. [Camp86] R. H. Campbell and B. Randell, ‘‘Error recovery in asynchronous systems,’’IEEE Transactions on Software Engineering, vol.SE-12, no.8, pp.811-826 (1986).

BIOGRAPHIES

A. Padma Priya presently holds the position of Assistant Professor in the department of Electronics and Communication Engineering at Nalla Malla Reddy Engineering College ,Hyderabad. I obtained B.Tech degree in the stream of Electronics and Communication Engineering from D.M.S S.V.H College of Engineering, Machilipatnam in 2008, M.Tech degree in the stream of VLSI & E.S from Sasi Institute of Engineering &Technology, Tadepalligudem in 2011. I published various papers in National and International Conferences .My current research interests are CPLD, Standard cells and Low power VLSI.

G.Rajesh Krishna presently holds the position of Assistant Professor in the department of Electronics and Communication Engineering at Nalla Malla Reddy Engineering College, Hyderabad. I obtained my Bachelors of Technology in Electronics and Communication Engineering from Alfa college of Engineering and Technology, Affiliated to JNTU, Anantapur in 2009,Master of Technology in Digital Systems and Computer Electronics from Rajeev Gandhi Memorial college of Engineering and Technology, Affiliated to JNTU, Anantapur in 2011.I published various

papers in International Journal of Engineering and Science Research .My current research interest are Wireless Communications, Image Processing & Low Power VLSI.