A Framework For Proactive Fault Tolerance[12]

Geoffroy Vallée
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37830
valleegr@ornl.gov

Kulathep Charoenpornwattana
Louisiana Tech University
Ruston, LA 71272, USA
kch020@coes.latech.edu

Christian Engelmann Anand Tikotekar
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37830
{engelmannc,tikotekaraa}@ornl.gov

Chokchai Leangsuksun
Louisiana Tech University
Ruston, LA 71272, USA
box@coes.latech.edu

Thomas Naughton
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37830
naughtont@ornl.gov

Stephen L. Scott
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37830
scottsl@ornl.gov

Abstract

Fault tolerance is a major concern to guarantee availability of critical services as well as application execution. Traditional approaches for fault tolerance include checkpoint/restart or duplication. However it is also possible to anticipate failures and proactively take action before failures occur in order to minimize failure impact on the system and application execution.

This document presents a proactive fault tolerance framework. This framework can use different proactive fault tolerance mechanisms, i.e.*, migration and pause/unpause. The framework also allows the implementation of new proactive fault tolerance policies thanks to a modular architecture. A first proactive fault tolerance policy has been implemented and preliminary experimentations have been done based on system-level virtualization and compared with results obtained by simulation.*

1 Introduction

Fault tolerance (FT) policies can typically be categorized into two sets: *reactive fault tolerance* policies and *proactive fault tolerance* policies. While reactive fault tolerance policies minimize the impact of failures on application execution when the failure effectively occurs; proactive fault tolerance policies aim at predicting failures and move running applications away from nodes that are predicted to fail.

The reactive approach is the solution which is most often selected by fault tolerant systems. Typically the application is periodically checkpointed and if a failure occurs, part of or the whole application is restarted. Because this approach is I/O intensive, we already know that this approach is not suitable for some classes of execution platforms, *e.g.*, large-scale systems [6].

Since proactive fault tolerance can predict failures, it is possible to move applications away for nodes that are about to fail. For instance [1] shows the interest of proactive fault tolerance for MPI applications, using Charm++ capabilities for the migration of MPI processes, migration and pause/unpause being two standard mechanisms for the implementation of proactive fault tolerance strategies. In [5] the authors show the benefit of using virtual machine (VM) migration for the implementation of proactive fault tolerance capabilities. However, these solutions directly implement a proactive fault tolerance policy into the system. These systems will therefore not be optimal for all execution platforms and applications.

In fact, the proactive fault tolerance approach creates a number of challenges: fault prediction (based on hardware monitoring mechanisms), prediction accuracy [9], implementation of a policy efficient for the target application and the target platform and so on. However, it is very rare to have the same set of monitoring capabilities on two different platforms. This heterogeneity and the fact that two different applications do not have the same constraints in terms of fault tolerance makes it impossible to have a "one-size-fits-all" proactive fault tolerance policy.

The intent of this document is not to study all the challenges associated with proactive FT, nor reactive FT. This paper focuses on a framework that eases the implementation of new proactive FT policies, based on abstractions of the underlying mechanisms and a set of modular components. The framework prototype is currently based on Xen [3]. However, the architecture is designed to be easily extended to other mechanisms, such as process migration and process pause/unpause. This framework, coupled to our fault tolerance simulator [8] provides a complete set of tools for the study of proactive fault tolerance policies.

The remainder of this paper is organized as follows: Section 2 presents the architecture of our framework and Section 3 presents a brief description of our first prototype which allows us to do preliminary experimentation; Section 4 concludes.

2 Proactive Fault Tolerance Framework

To be used on High-Performance Computing (HPC) systems, our framework has to respect the following constraints: (i) the overhead created by the system has to be minimized when the system is healthy, (ii) the framework has to be scalable in order to be used on modern large-scale computing platforms, and (iii) the system is composed of different kinds of node, from compute nodes for application execution to service nodes which are specialized to offer a specific service (*e.g.* I/O nodes). Since we target mainly computationally intensive applications, we avoid any extra computation on compute nodes; framework operations are limited to the service node(s).

For simplification, we use the concept of a *control node* which is used to implement the central component of our framework. The actual implementation may not be on a single node; this is only a logical view. The implementation of the *controller* can be done via distributed processes working together or a single process with some duplication mechanisms in order to avoid the creation of a single point of failure. For instance a solution for symmetric active-active replication can be used across multiple physical nodes offering more than 99% of availability [4]. Networking techniques such as MRNet [7] may also be used for communication between our controller and compute nodes. Such techniques allow one to designate few nodes for networking operations and are in charge of managing communications in a scalable way. However, the techniques used for the implementation of the control node are beyond the scope of this document, since the implementation of scalable runtime systems for HPC is still a very active research topic.

We select an event based architecture: information are filtered locally on compute nodes and if an abnormal behavior is detected, an alarm (*i.e.*, an event) is sent to the control node. Then the control node analyzes the alarm and may decide to proactively act to avoid a failure. This avoids periodic polling and diffusion of data: only information from local nodes are periodically extracted from hardware probes but when the system is healthy, no events, *i.e.*, no communications or extra computation, is generated.

Since the solution presented in this document is designed to be generic and flexible, different underlying mechanisms may be used. These mechanisms may be from a system-level virtualization solution (using VM pause/unpause and migration) to solutions providing process-level pause/unpause and migration. The proactive FT mechanisms can then deal with different granularity of computing entities: from processes to VMs. We use the generic term *computing entity* in this document to refer to processes or VMs or so.

2.1 Proactive Fault Tolerance Mechanisms

To implement a proactive fault tolerance policy several basic mechanisms are needed. First, we need to get data from the system for failure prediction. When a failure or an abnormal behavior is predicted, the fault tolerance policy has to be activated and an action (such as application pause or application migration) may occur.

Therefore our mechanisms are composed of three main *components*: (i) the *fault predictor*, (ii) the *policy daemon*, and (iii) the *fault tolerance daemon* (see Figure 1). Each

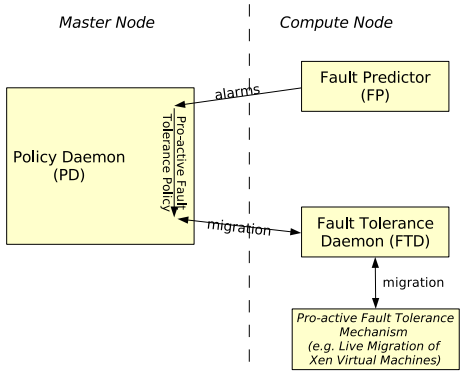

Figure 1. Architecture of the Fault Tolerance Framework

of these components is based on our event system which

[1]ORNL's work was supported by the U.S. Department of Energy, under Contract DE-AC05-00OR22725.

[2]Part of this work has been completed as part of Google Summer of Code 2007 project for Open Source Cluster Application Resources (OSCAR) under project name "Cluster Virtualization with Xen" mentored by Geoffroy R. Vallee and presented in poster session at Oklahoma Supercomputing Symposium 2007, University of Oklahoma, Norman, OK

allows asynchronous communications and do not compromise the framework in the case of failures.

Event System In order to minimize the communications in the system, we have designed a subsystem called *event system* to handle all communications inside our framework. The event system is considered as a core of the framework and provides an abstraction for communications between other framework components. It also allows us to easily use any kind of communication system: from the usage of sockets to the usage of overlay network such as MRNet. The central concept of the event system is the concept of a *mailbox*, which eliminates repeated global data polling. Three parties are involved in the mailbox system: i) subscribers, ii) publisher, and iii) a mailbox manager. The mailbox manager creates a new mailbox; other components of the system can register to the system as a publisher or a subscriber. When an event is posted, the mailbox manager tries to deliver (asynchronously) the event to subscribers. Any component in the system could be a subscriber and/or a publisher. Doing so all communications are asynchronous and even in the case of a failure, the entire system is not compromised. Applying this technique to our framework ensures that no additional communication is generated between nodes when no events are generated (*e.g.*, in a healthy system). Moreover, the mailbox manager is not implemented on compute nodes, the computation generated by the mailbox handling does not impact the application execution.

Fault Predictor The *fault predictor* (FP) runs on each compute node and filters local information to predict failures based on system data. The data analysis allows us to prevent a global polling approach for the implementation of the framework: events are generated by the FP only if an abnormal behavior is observed. Modern operating systems such as Linux include systems monitoring mechanisms based on hardware probes. Data from these probes can be used for fault prediction. Currently the FP analyzes system logs to detect disk errors and also analyzes hardware sensors using lm-sensor data to detect aberrant system temperature. If such issues are detected, an alarm is sent to the *policy daemon* on the master-node. We currently assume information taken from hardware monitors is accurate and does not lead to false prediction.

Policy Daemon The *policy daemon* (PD) is composed of two parts: a trigger that receives alarms from the compute nodes, and the fault-tolerance policy. When an alarm is received by the trigger, this alarm is identified by the alarm type and the identifier of the alarm source, an *internal event* is then created and sent to the fault-tolerance policy: the event activates the policy and according to the system status, the event is analyzed. Since FPs are based on low-level monitoring mechanisms which may have different behavior depending on available hardware sensors, we provide a tool for the creation of new policies (see Section 2.2).

Fault Tolerance Daemon Thus the *fault tolerance daemon* (FTD) implements the fault tolerance protocol on the compute node side, presented in Section 2.1.1. The FTD triggers order of action from the PD. Since our framework is based on only two mechanisms, pause/unpause and migration (for a VM [2] or a process), the FTD can only perform computing entity migration, pause or unpause. However, these actions differ in function of the underlying mechanisms. For instance, if users do not want to modify their OS and their application, it is possible to use Xen (the fault tolerance granularity is then a VM); if users accept to modify their OS, it is possible to use systems such as Kerrighed which provides proactive fault tolerance mechanisms at the process level by modifying the Linux kernel. For that, we include the concept of *connector* which is an abstraction of underlying mechanisms such as tools for OS virtualization. Therefore the concept of connector allows us to provide a generic solution for proactive fault tolerance since different low-level mechanisms may be plugged within our framework (a connector can be considered as a plug-in in our framework).

2.1.1 Fault Tolerance Protocol

When a failure is predicted and if the proactive fault tolerance policy decides to preemptively migration or pause a computing entity, the policy daemon (PD) has to communicate with the fault tolerance daemons (FTD) which are running on the compute nodes. Even though we do not address reactive fault tolerance in this document, we have to be sure that communications between the PD and the FTD end on a coherent view of the system. For instance, if a node fails during the communication between the two daemons, the PD has to detect it and notify the system administrator that the status of the system is not coherent for proactive fault tolerance.

Therefore, a *negotiation* between the PD and the FTD takes place. Figure 2 presents the graph representing the negotiation protocol when no errors occurs. If one error occurs during the negotiation, *i.e.*, if one or several nodes cannot initiate the negotiation or cannot perform the proactive fault tolerance action, the negotiation is canceled. If this happens during a negotiation for migration, the migration is canceled; if this happens during a negotiation for pause, computing entities already paused during the negotiation are unpaused (through a new negotiation); and if it is not then possible to fall back to a stable state (*i.e.*, the

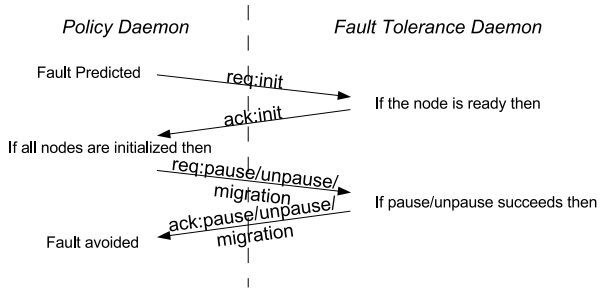

Figure 2. Protocol for Proactive Fault Tolerance in an Healthy System

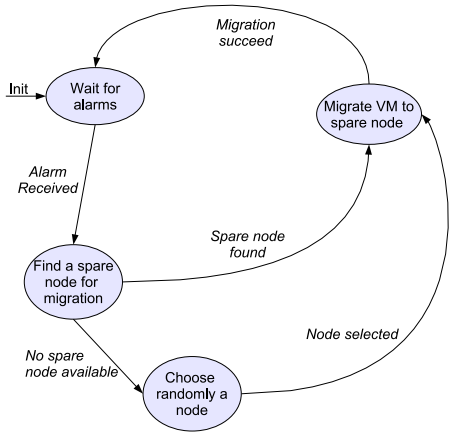

Figure 3. Example of a Finite-State Machine for the Specification of a Proactive Fault Tolerance Policy

state of a computing entity changed during the negotiation, an error occurs but it is not possible to revert the negotiation), it is considered as a failure therefore the proactive fault tolerance policy daemon stops and notifies the system's administrator.

This protocol allows the framework to perform proactive fault tolerance actions but is also tolerant to faults in the sense that failures are detected and the framework stops in order to avoid incoherent decisions.

2.2 Proactive Fault Tolerance Policies Manager

A proactive fault tolerance policy has to address several questions: *When* do I have to do something? *Who* am I supposed to migrate or pause? and *How* can I do that? A way to implement proactive fault tolerance policies is therefore to implement an algorithm that answers these three questions.

A simple solution for that is to specify the policy via a finite-state machine: each event leads to a new state and the different phases of policy algorithm can also be viewed as a state (for instance each of the three questions represents different states). Figure 3 gives an example of such a policy where every time an alaram is received from a remote node, the computing entities are moved away from that node using spare nodes; when spare nodes are no longer available, a healthy node is randomly selected for the migration, *i.e.*, VMs are stacked.

In order to ease the development of new FT policies, we provide a software developer kit (SDK). This SDK allows developers direct access to data and features of the PD such as the event trigger and the negotiation mechanism to communicate with FTD on compute nodes. Since we assume the policy may be defined via a finite-state machine, we provide a set of Python APIs and classes that may be used to define or tune each framework components. A high-level language for the specification of a finite-state machine is also available, based on XML: an XML document allows one to define each framework component and a XSL-T document generates the associated policy code (code based on the Python API).

3 Evaluation

The validation of our framework is not actually a simple task. Some may argue that it can be done by the implementation of several policies but we think that such an approach is not valid since it is lacking a reference to compare to. Instead we decided to compare the results of the implementation of a specific proactive fault tolerance policy with the results of the same policy obtained by simulation. Our simulator [8] is based on the LLNL's ASCI white system system logs and our experimentation platform consists of a 40 nodes cluster. Each physical node has a single Xen VM having 250MB of memory (the number of VM is explicitly specified if VMs are stacked on physical machines); host OSes have 200MB of memory.

We conducted two sets of experiments: using (i) 16 nodes, and (ii) 32 nodes. We also use two different configurations of the HPCC benchmark application in order to keep the application execution time as constant as possible: the HPCC problem size is 9000 when we use 16 nodes, whereas the problem size is 10700 for 32 nodes. The increase in problem size was based on a trial and error method. Note that the number of nodes and processes increase the amount of time spent waiting for messages and synchronization. Therefore the execution time does not increase proportionally to the problem size.

The current framework has been implemented for small-size to middle-size system. This choice is driven by the fact that we validate the framework comparing the im-

pact of a specific policy using our framework and our simulator. Moreover, the implementation of a scalable runtime system is still a very active research topic and the focus of this paper is not to address scalability issues but only to present a proof-of-concept prototype. However, the design has been done with scalability in mind; scalability being one of our next points of study. Based on this assumptions, the framework is composed of n fault predictors (one on each compute node), and one policy daemon and one fault tolerance daemon, located on a single physical node (our local control node is in fact implemented on a single node). The current implementation uses one service node, which is an acknowledged single point of failure.

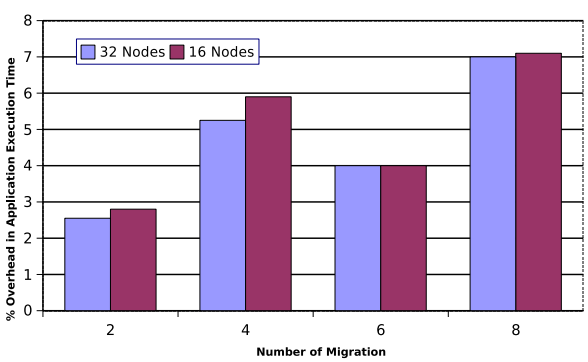

Figure 4. Live Migration Overhead

We implemented the policy presented in Figure 3: if the PD receives alarms from a remote node (which are generated based on the alarm pattern extracted from the system logs used with our simulator), the VM(s) is migrated away from that node using a spare node. If no spare nodes are available, VMs are stacked on healthy nodes. We assume wrong failure prediction does not occur in order to simplify our model.

3.1 Evaluation Via Experimentation

To be able to compare experimental results and results from our simulator, we first evaluate the overhead created by the VM migration using Xen 3.0.2. Figure 4 shows the overhead in application execution time when a VM is live migrated to another physical node. The above experiments were carried out with an application running on 16 nodes and 32 nodes and having memory footprint of 45 MB and 30MB respectively on each node. The overhead is quite small even after migrating 8 times. Moreover, there is not much difference in the application execution overhead between the two sets of results. This is expected since Xen has a very low downtime when live migrating a VM; the overhead is quite small even after migrating 8 times (between 2.5 and 7%). However the time to migrate may

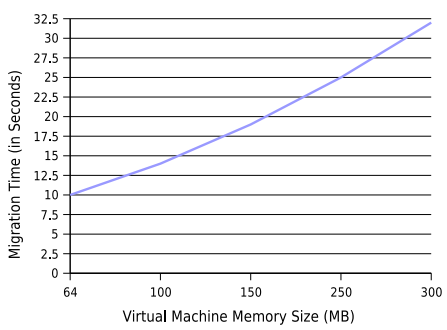

Figure 5. Impact of Virtual Machine Footprint on the Migration Cost

change depending upon the hardware characteristics and the VM's memory footprint [2]; Figure 5 shows the "time to migrate" as a function of the VM's memory footprint using our cluster.

Figure 6 describes the results when spare nodes are already used (all physical nodes have at least one VM running) and a live migration has to be done for proactive fault avoidance (therefore several VMs have to be stacked on a single nodes). The effect of stacking even one VM is

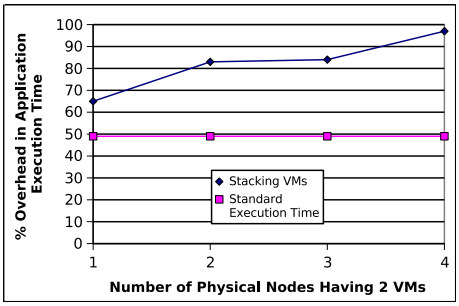

Figure 6. The Effect of VM Stacking on Application Execution Time

staggering. The fact that the overhead does not increase proportionately is due to the parallelism of the application.

3.2 Comparison With Simulation Results

We previously developed a simulator for fault tolerant systems which allows us to study the behavior of different policies modifying the algorithm for FT but also parameters of each mechanisms used by the FT policy and the failure characteristics of the simulated platform (number of failures, number of predicted failures and so on). For instance, it is possible to specify the overhead of a VM migration and the number and frequency of predicted fail-

ures. Based on these logs, the simulator typically reproduce a simular behavior for the simulated application lifetime, *i.e.*, the simualtor produces events based on system logs (typically failures) and evaluates the impact of the application execution, taking into account the overhead of the different fault tolerance mechanisms specified.

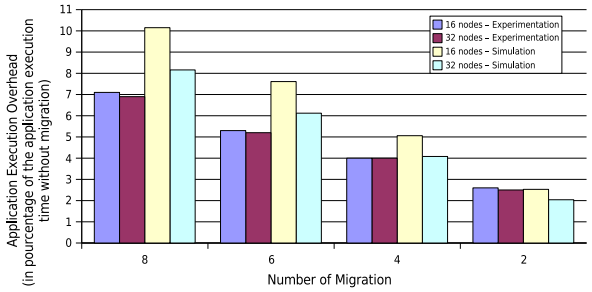

Figure 7. Comparison of Experimental Results and Results by Simulation

Figure 7 shows a comparison of the two sets of results: the first set evaluating the proactive fault tolerance using our framework on a real platform, and the second set simulating the platform. We can see that the results from the simulator are close to the results using our framework. The maximum deviation is 3% for 16 nodes. The deviation between the simulation and experimentations is mostly due to the fact that in the simulator, the migration cost is constant whereas on our platform the migration overhead differs as a function of the number of node and the memory footprint of each VM. However, a 3% deviation is acceptable and shows that our framework has an expected behavior, *i.e.*, does not introduce significant overhead.

4 Conclusion

The usage of proactive fault tolerance policies creates multiple challenges: fault prediction, prediction accuracy, implementation of an efficient policy for a specific set of requirements and so on. The effort presented in this document does not aim to address all these issues; it only addresses issues for the implementation of proactive fault tolerance policies for a given set of requirements (for instance from execution platform and application characteristics). For that, our framework is based on a modular architecture which allows one to plug different proactive fault tolerance mechanisms (*e.g.*, VM or process migration) and to define different policies. Such a framework enables the implementation of various fault tolerance policies, including policies presented in the literature that were not validated by experimentation; therefore our framework, coupled with our fault tolerance simulator, provides a complete solution for the study of proactive fault tolerance policies. The framework prototype currently provides a single policy based on Xen VM migration but new policies are currently under development.

Other challenges associated with proactive fault tolerance are currently under investigation: the study of the prediction accuracy when using modern hardware monitoring probes; the analysis of failure logs for large-scale systems over a long period of time; the improvement of our simulator, and the implementation of a scalable runtime for our framework (to target large-scale HPC systems). We are also extending our framework for the support of reactive fault tolerance policies.

References

[1] S. Chakravorty, C. Mendes, and L. Kale. Proactive fault tolerance in large systems. *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11). IEEE Computer Society*, 2005.

[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, Boston, MA, May 2005.

[3] E. Dragovic, P. Barham, K. Fraser, S. Hand, T. H. A. Ho, R. Neugebauery, I. Pratt, and A. Warfield. Xen and the art of virtualization. *In the Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[4] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He. Symmetric active/active high availability for high-performance computing system services. *Journal of Computers (JCP)*, 1(8):43–54, 2006.

[5] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 23–32, New York, NY, USA, 2007. ACM Press.

[6] R. Oldfield. Investigating lightweight storage and overlay networks for fault tolerance. In *HAPCW'06: High Availability and Performance Computing Workshop*, Santa Fe, New Mexico, USA, OCT 2006.

[7] P. C. Roth, D. C. Arnold, and B. P. Miller. Mrnet: A software-based multicast/reduction network for scalable tools. In *SuperComuting Conference 2003*, Phoenix, Arizona, Nov. 2003.

[8] A. Tikotekar, G. Vallée, T. Naughton, S. L. Scott, and C. Leangsuksun. Evaluation of fault-tolerant policies using simulation. In *Proceedings of the* 9^{th} *IEEE International Conference on Cluster Computing (Cluster)*, Austin, Texas, USA, Sept. 17-20, 2007.

[9] D. Turnbull and N. Alldrin. Failure prediction in hardware systems.