Universitat Autònoma de Barcelona

Fault Tolerance Configuration and Management for HPC applications using RADIC Architecture

Jorge Luis Villamayor Leguizamón

Universitat Autònoma
de Barcelona

Escola d'Enginyeria.
Departament d'Arquitectura de
Computadors i Sistemes Operatius

Fault Tolerance Configuration and Management for HPC applications using RADIC Architecture

Thesis submitted by ***Jorge Luis Villamayor Leguizamón*** for the degree of Philosophiæ Doctor by the Universitat Autònoma de Barcelona, under the supervision of the ***Dr. Dolores Isabel Rexachs del Rosario***, done at the Computer Architecture and Operating Systems Department, PhD. in Computer Science.

Barcelona, September, 2018

Fault Tolerance Configuration and Management for HPC applications using RADIC Architecture

Thesis submitted by ***Jorge Luis Villamayor Leguizamón*** for the degree of Philosophiæ Doctor by the Universitat Autònoma de Barcelona, under the supervision of the ***Dr. Dolores Isabel Rexachs del Rosario***, done at the Computer Architecture and Operating Systems Department, PhD. in Computer Science.

Barcelona, September, 2018

Dr. Dolores Isabel Rexachs del Rosario	**Jorge Luis Villamayor Leguizamón**
Thesis Advisor	Thesis Author

I would like to dedicate this thesis to God, this was only possible with your unconditional and invisible guidance.

When I felt lost and in the dark, there was always a light shining for me through this difficult ride.
to you, my lovely wife, Laura Espínola.

Lastly, to my parents, who have given me the opportunity of an education throughout my life and for their constant support despite the distance.

Acknowledgements

First, I want to express my deepest gratitude to my supervisor Dolores Rexachs, for her guidance during this journey. Thanks for giving me the insights of this thesis topic, and helping me step by step to make this work a real success. I would like to recognize Emilio Luque for the knowledge conveyed during these years of research.

To my fellow-mates, colleges, and friends in the lab, they make this endeavor seems smoother. I must also thank all the staff members of the Computer Architecture and Operating Systems Department (CAOS), for always being so kind, gentle and helpful.

I would like to acknowledge Nokia Bell Labs, for opportunity to be part of their research team during my stay in Ireland. My abroad experience, was one of the best experiences during this Ph.D. I will never forget the people's kindness and hospitality. During this experience, they show me how professional research is done. Special appreciation to Diego Lugones, for his ideas, contributions, and support during the stay. I can not omit to mention Daniel Franco, who managed to present us: Thank you.

A recognition to the Facultad Politécnica of the Universidad Nacional de Asunción, my *alma mater*, for acting as a connector by offering information of scholarships to continue with my education. Special acknowledgements to my colleges at the Informatics Department.

I want to sincerely thank you, my loving wife, Laura Espínola. For always giving me love, strength, patience, wisdom and keeping my faith alive during the years of this Ph.D. As you told me from the beginning of this adventure: "*We will extend the boundaries of human knowledge*", and as always, you were right.

To my parents and brothers, they are the reason of what I became today. For their care and support during this journey. I will always be in debt with you mom and dad.

Lastly, thanks to my colleges and friends from all over the world.

Jorge Villamayor
Barcelona, September, 2018.

Abstract

High Performance Computing (HPC) systems continue growing exponentially in terms of components quantity and density to achieve demanding computational power. At the same time, cloud computing is becoming popular, as key features such as *scalability*, *pay-per-use* and *availability* continue to evolve. It is also becoming a competitive platform for running parallel HPC applications due to the increasing performance of virtualized, highly-available instances. Although, augmenting the amount of components to create larger systems tends to increment the frequency of failures in both clusters and cloud environments. Nowadays, HPC systems have a failure rate of around 1000 per year, meaning a failure every approximately 8 hours.

Most of the parallel distributed applications are built on top of a Message Passing Interface (MPI). MPI implementations follow a default *fail-stop* semantic, which aborts the execution in case of host failure in a cluster. In this case, the application owner needs to restart the execution, which affects the wall clock time and, also, the cost since it requires to acquire computing resources for longer periods of time.

Fault Tolerance (FT) techniques need to be applied to MPI parallel executions in both, cluster and cloud environments. With FT techniques, high availability is ensured for parallel applications. In order to apply some FT solutions, administrator privileges are required, to install them in the cluster nodes. Moreover, when failures appear human intervention is required to recover the application. A solution, which minimizes users and administrators intervention is preferred.

Rollback-Recovery protocols represent a fundamental component to implement FT techniques. The protocols consist of snapshots created from the parallel execution and stored as checkpoints. In case of failures, the application can be recovered using the stored checkpoints. Coordinated, uncoordinated and semi-coordinated checkpoints are some of the most used protocols.

A contribution of this thesis is a Fault Tolerance Manager (FTM) for coordinated checkpoint, which provides the application's users with automatic recovery from failures when losing computing nodes. It takes advantage of node local storage to save checkpoints, and it distributes copies of them along all the computation nodes,

avoiding the bottleneck of a central stable storage. We also leverage the FTM to use uncoordinated and semi-coordinated rollback recovery protocols. In this contribution, FTM is implemented in the application-layer. Furthermore, a dynamic resource controller is added to the FTM, which monitors the FT protection resource usage and performs actions to maintain an acceptable level of protection.

Another contribution aims to the FT protection and recovery tasks configuration. Two models are introduced. The First Protection Point model (FPP) determines the starting point to introduce FT protection gaining benefits in terms of total execution time including failures, by removing unnecessary checkpoints. The second model allows improving the FT resource configuration for the recovery task.

Regarding cloud environments, we propose Resilience as a Service (RaaS), a fault tolerant framework for HPC applications, which uses FTM. RaaS provides clouds with a highly available, distributed and scalable fault-tolerant service. It redesigns traditional HPC protection and recovery mechanisms, to natively leverage cloud capabilities and its multiple alternatives for implementing FT tasks.

To summarize, this thesis contributes on providing a Multi-platform Resilience Manager (MRM), suitable for traditional bare-metal clusters and clouds (public and private). The presented solution provides FT in an automatic, distributed and transparent manner in the application and user levels according to the users, applications, and runtime requirements. It gives the users critical FT information, allowing them to trade-off cost and protection keeping the mean time to repair within acceptable ranges.

Several experimental environments such as bare-metal clusters and cloud (public and private), running different parallel applications were used during the experimental validations. The experiments verify the functionality and improvement of the contributions. Moreover, they also show that the Mean Time To Repair is bounded within acceptable ranges.

Keywords: Fault Tolerance, High Performance Computing, Cloud Computing, Rollback-Recovery Protocols, Checkpoint/Restart.

Resumen

Los sistemas de computación de alto rendimiento (HPC) continúan creciendo exponencialmente en términos de cantidad y densidad de componentes para lograr mayor potencia de cálculo. Al mismo tiempo, *cloud computing* se está volviendo popular, ya que las características clave tales como *escalabilidad*, *pay-per-use* y *alta disponibilidad* continúan evolucionando. También se está convirtiendo en una plataforma competitiva para ejecutar aplicaciones paralelas HPC debido al rendimiento cada vez mayor de instancias virtualizadas y de alta disponibilidad. Sin embargo, aumentar la cantidad de componentes para crear sistemas más grandes tiende a incrementar la frecuencia de fallos tanto en clústeres como en cloud. Hoy en día, los sistemas HPC tienen una tasa de fallos de alrededor de 1000 por año, lo que significa un fallo cada aproximadamente 8 horas.

La mayoría de las aplicaciones paralelas distribuidas se construyen sobre una interfaz de paso de mensajes (MPI). Las implementaciones MPI siguen una semántica *fail-stop* predeterminada, que aborta la ejecución en caso de fallos de host en un clúster. En este caso, el propietario de la aplicación debe reiniciar la ejecución, lo que afecta el tiempo total esperado del mismo y, además, el costo, ya que requiere adquirir más recursos durante periodos de tiempo más largos.

Las técnicas de Tolerancia a Fallos (TF) deben aplicarse a las ejecuciones paralelas de MPI tanto en clúster como en cloud. Con las técnicas de TF, se garantiza una alta disponibilidad para aplicaciones paralelas. Para aplicar algunas soluciones de TF, se requieren privilegios de administrador para instalarlas en los nodos del clúster. Además, cuando aparecen fallos, se requiere intervención humana para reiniciar la aplicación. Se prefiere una solución que minimice la intervención de usuarios y administradores.

Los protocolos rollback-recovery representan un componente fundamental para implementar las técnicas de TF. Los protocolos consisten en almacenar el estado de la ejecución de una aplicación paralela en *checkpoints*. En caso de fallos, la aplicación se puede recuperar utilizando los *checkpoints* almacenados. Los protocolos de checkpoints coordinados, no-coordinados y semi-coordinados son algunos de los más utilizados.

Una contribución de esta tesis es un *Fault Tolerance Manager* (FTM) para checkpoints coordinados, que proporciona a los usuarios la recuperación automática de fallos al perder nodos de clúster. FTM aprovecha el almacenamiento local en los nodos para guardar los checkpoints, y distribuye sus copias entre los nodos de computo, evitando el cuello de botella de un almacenamiento centralizado.

También aprovechamos FTM para utilizar protocolos de rollback-recovery no-coordinados y semi-coordinados. En esta contribución, FTM se implementa en la capa

de aplicación. Además, se agrega un controlador dinámico de recursos al FTM, que monitoriza el uso de recursos destinados para la protección FT y realiza acciones para mantener un nivel aceptable de protección.

Otra contribución apunta a la configuración de tareas de protección y recuperación de la TF. Se presentan dos modelos: el modelo *First Protection Point* (FPP) determina el punto de partida para introducir la protección de TF de modo a obtener beneficios en términos de tiempo total de ejecución, incluyendo fallos, eliminando checkpoints innecesarios. El segundo modelo permite mejorar la configuración de recursos de la TF para la tarea de recuperación.

Con respecto a entornos cloud, proponemos Resilience as a Service (RaaS), un servicio que provee TF para aplicaciones HPC, que utiliza FTM. RaaS proporciona al cloud un servicio de TF distribuido, escalable y altamente disponible. Rediseña los mecanismos tradicionales de protección y recuperación de HPC, para aprovechar de forma nativa las capacidades del cloud y sus múltiples alternativas para implementar tareas de TF.

En resumen, esta tesis contribuye a proporcionar un gestor de tolerancia a fallos multi-plataforma "Multi-Platform Resilience Manager" (MRM), adecuado para entornos de clústers y clouds (públicos y privados). La solución presentada proporciona TF de forma automática, distribuida y transparente en las capas de aplicación y usuario según los requisitos de usuarios, aplicaciones y entorno de ejecución. Brinda además, a los usuarios información crítica de la TF, lo que les permite compensar entre costos y protección, manteniendo el tiempo medio de reparación dentro de rangos aceptables.

Durante las validaciones experimentales se utilizaron varios entornos experimentales, como clústeres tradicionales y cloud (públicos y privados), ejecutando diferentes aplicaciones paralelas. Los experimentos verifican la funcionalidad y la mejora de las contribuciones. Además, también muestran que el tiempo medio de reparación se encuentra limitado y dentro de rangos aceptables.

Keywords: Tolerancia a Fallos, Computación de Altas Prestaciones, Cloud Computing, protocolos Rollback-Recovery, Checkpoint/Restart.

Resum

Els sistemes de computació d'alt rendiment (HPC) continuen creixent exponencialment en termes de quantitat i densitat de components per aconseguir un poder computacional exigent. Al mateix temps, la informàtica en núvol s'està convertint en popular, ja que les funcions clau com textit scalability, textit pay-per-use i textit availability continuen evolucionant. També s'està convertint en una plataforma competitiva per executar aplicacions HPC paral · leles a causa de l'augment del rendiment d'instàncies virtualitzades i altament disponibles. Encara que, augmentar la quantitat de components per crear sistemes més grans tendeix a augmentar la freqüència d'errors en ambdós clústers i entorns en el núvol. Avui en dia, els sistemes HPC tenen una taxa de fracàs de prop de 1000 per any, que significa un fracàs cada 8 hores aproximadament.

La majoria de les aplicacions distribuïdes en paral · lel es construeixen a la part superior d'una interfície de pas de missatges (MPI). Les implementacions de MPI segueixen un semàntic textit fail-stop per defecte, que evita l'execució en cas d'error del servidor en un clúster. En aquest cas, el propietari de l'aplicació necessita reiniciar l'execució, que afecta el temps del rellotge de paret i, a més, el cost, ja que requereix l'adquisició de recursos informàtics durant períodes de temps més llargs.

Les tècniques de tolerància a fallades (FT) s'han d'aplicar a les execucions paral · leles de MPI en ambients tant en clúster com en núvol. Amb les tècniques FT, es garanteix una alta disponibilitat per a aplicacions paral · leles. Per aplicar algunes solucions FT, es requereixen privilegis d'administrador, per instal · lar-los en els nodes del clúster. A més, quan es produeixi un error, es requereix una intervenció humana per recuperar la sol · licitud. Es prefereix una solució que minimitzi la intervenció dels usuaris i administradors.

Els protocols de recuperació-recuperació representen un component fonamental per implementar tècniques de FT. Els protocols consisteixen en instantànies creades a partir de l'execució paral · lela i emmagatzemades com a punts de control. En cas de fallades, es pot recuperar l'aplicació mitjançant els punts de control emmagatzemats. Els punts de control coordinats, descoordinats i semi-coordinats són alguns dels protocols més utilitzats.

Una contribució d'aquesta tesi és un gestor de tolerància a falles (FTM) per al punt de control coordinat, que proporciona als usuaris de l'aplicació una recuperació automàtica de fallades al perdre nodes informàtics. Aprofita l'emmagatzematge local del node per guardar els punts de control i distribueix còpies d'ells al llarg de tots els nodes de còmput, evitant el coll d'ampolla d'un emmagatzematge estable central.

També aprofitem l'FTM per utilitzar protocols de recuperació de descompte no coordinats i semi-coordinats. En aquesta contribució, FTM s'implementa en la capa d'aplicació. A més, s'afegeix un controlador de recursos dinàmics a la FTM, que supervisa l'ús de recursos de protecció FT i realitza accions per mantenir un nivell de protecció acceptable.

Una altra contribució apunta a la configuració de les tasques de protecció i recuperació de FT. S'introdueixen dos models. El model First Point de protecció (FPP) determina el punt de partida per introduir la protecció FT guanyant beneficis en termes de temps total d'execució, incloent fracassos, eliminant punts de control innecessaris. El segon model permet millorar la configuració de recursos FT per a la tasca de recuperació.

Pel que fa als entorns cloud, proposem Resilience as a Service (RaaS), un marc tolerant a fallades per a aplicacions HPC, que utilitza FTM. RaaS proporciona núvols amb un servei altament disponible, distribuït i escalable i tolerant a les falles. Redissenya els mecanismes tradicionals de protecció i recuperació de l'HPC, per potenciar nativament les capacitats del núvol i les seves múltiples alternatives per implementar tasques de FT.

Per resumir, aquesta tesi contribueix a proporcionar un gestor de resistència multiplataforma (MRM), adequat per a clústers i núvols (públics i privats). La solució presentada proporciona FT de manera automàtica, distribuïda i transparent en els nivells d'aplicació i usuari d'acord amb els requisits dels usuaris, aplicacions i temps d'execució. Ofereix als usuaris informació crítica sobre FT, que els permet reduir el cost i la protecció mantenint el temps mitjà de reparar dins de rangs acceptables.

Es van utilitzar diversos entorns experimentals com els clústers de metall nu i el núvol (públic i privat), que van executar diferents aplicacions paral · leles durant les validacions experimentals. Els experiments verifiquen la funcionalitat i millora de les contribucions. D'altra banda, també mostren que el temps mitjà per reparar es limita dins de rangs acceptables.

Keywords: Tolerància a Fallos, Computació d'Altes prestacions, Cloud Computing, protocols Rollback-Recovery, Checkpoint/Restart.

Table of contents

List of figures

List of tables

Nomenclature

Acronyms / Abbreviations

BLCR Berkeley Lab Checkpoint/Restart

CD Checkpointing Daemon

DMTCP Distributed MultiThreaded CheckPointing

FPP First Protection Point Model

FT Fault Tolerance

FTM Fault Tolerance Manager

HPC High Performance Computing

MPI Message Passing Interface

MRM Multi-platform Resilience Manager

MTBF Mean Time Between Failures

MTTR Mean Time To Repair

RaaS Resilience as a Service

RADIC Redundant Array of Distributed Independent Controllers

SDC Silent Data Corruption

SNCM Spare Node Configuration Model

ULFM User-Level Failure Mitigation

Chapter 1

Introduction

"The only truth wisdom is in knowing you know nothing."

Socrates

Computers are one of the most advanced creation of the human kind. They basically started a revolution in the way people think and accomplish their day to day tasks. Nowadays computer's systems are present everywhere and in everything of our daily life, i.e: medical devices, industrial plants, home appliance devices, air crafts, automobiles, among others.

These systems are processing great amount of electronic transactions, market operations, without even counting the quantity of information that is stored. With this increasing demand, it was necessary to create a paradigm, where the usage of clusters of computers improves the throughput of computing power [13, 18]. Along with the paradigm of providing clusters, the parallel computing is created with the idea of dividing problems into several tasks, in order to solve them using separated processes in parallel manner.

Clusters are commonly used to deliver High Performance Computing (HPC). Most of the parallel applications, which runs on HPC systems, are implemented using Message Passing Interface (MPI) that is used to enable interprocess communications involved in a parallel computation. The MPI interface is widely used and has become a *de facto* standard [36].

For long time, large physical clusters were the main resource to execute HPC for both industrial and academic sectors. Although another disruption appeared with Cloud Computing, which drastically affected how sectors are acquiring computational power to run their HPC applications. The evolution in terms of performance, together

with the multiplicity of pricing models, elasticity and high-availability make cloud a very competitive platform for scientific computing and HPC vertical markets in general [42]. However, moving HPC solutions to cloud is challenging, particularly in guaranteeing the completion of parallel, long-running, stateful applications.

Both physical and virtual clusters are built with bare-metal hosts. In order to keep up with the user's demand for HPC, manufactures have been increasing the density and quantity of components. The constantly increasing scale of HPC platforms tends to increment the frequency of hard failures in clusters and cloud environments. Contemporary HPC systems the Mean Time Between Failures (MTBF) is in range of few hours, depending on the maturity and age of installation [78, 81]. Furthermore, requirements for low power consumption, are lowering the operation frequency of CPUs, causing even more failures probabilities [25, 29].

When a hard failure occurs on a cluster node during a MPI parallel application execution, its is abruptly stopped. This is due mostly because the MPI implementations follow a *fail-stop* semantic, which aborts executions in presence of failures. In consequence users need to restart the execution, which affects the wall clock time of expected execution results. The repair or replace resource, the restart time, and the re-execution time are not a despreciable amount of time. Moreover, the need for human intervention in order to perform the verification against failures, and mitigation actions when required [11, 17].

Considering the previous scenario, Fault Tolerance (FT) techniques needs to be applied to MPI parallel executions in both, cluster and cloud environments. With FT techniques high availability is ensured for parallel applications. One of the issues when implementing FT is the cost in terms effort users have to make its application resilient to failures. Furthermore, in order to apply some FT solutions, administrator privileges are required, to install them in the cluster nodes. A solution, which minimizes users and administrators is preferred.

Rollback-Recovery protocols are mostly used within FT solutions. They basically consist on saving the state of the parallel execution as checkpoint. In case of failures, the application can be recovered using the most recent checkpoint. Applying FT to parallel executions comes with a considerable cost in terms of resources and overhead [30].

A well-known challenge arises, which is the configuration of the frequency in which checkpoints are taken during the application execution. It actually determines how much overhead is added to the failure-free execution. Collateral factors also influence the overhead, such as the storage resources performance and availability that is used

to persist the above mentioned checkpoints. For a checkpoint based FT, avoiding not necessary checkpoints can help reduce the failure-free execution overhead.

When a failure affects a computation node during a parallel execution, FT mechanisms, usually uses spare node resource to re-configure the execution environment and resume the application execution. Sometimes, spare nodes are not configured due to users or runtime requirements. An issue in the fault tolerance topic is to determine if an application can continue its execution when no spare resource is available [61].

Coordinated, uncoordinated and semi-coordinated are some of the most known protocols [17][30]. The coordinated protocol synchronizes the application processes to create a consistent state. Meanwhile, the uncoordinated protocol, enables the processes take checkpoints independently, when it is more convenient. Although, it is combined with an event logging facility, to store interprocess exchanged messages used when restoring failed processes [32, 66]. The semi-coordinated, combines both protocols, coordinated and uncoordinated, performing a coordination between processes inside a computation node, and storing exchanged messages among the processes in different computation nodes.

Applying a specific protocol would be dependent on the application requirements. For instance, coordinated protocols are widely used for tightly coupled applications and with moderate number of processes. Meanwhile, having a large number of application processes may incur a source of overhead, due to the coordination operation, hence the uncoordinated would be a suitable protocol. For large multi-core systems, or multi-site virtual clusters, the semi-coordinated, can enable the local coordination of processes, avoiding to persist local exchanged messages. An issue is to offer multiple Rollback-Recovery protocol alternatives, to better suit the specific execution requirement.

Taking into consideration the presented matters, following challenges are approached by this thesis:

- Provide a multi-platform fault tolerance architecture to offer high availability for parallel message passing applications execution in bare-metal clusters and cloud environments.

- Re-design traditional FT solutions for bare-metal clusters, which usually are designed to operate on a reduced and static number of physical resources, to make use of the diversity and on-demand facilities that cloud offers to acquire resources regarding computation and storage.

- Offer a FT solution which is independent of the operative system, and the MPI library implementation with the well-known rollback-recovery protocols, to give multiple configuration alternatives that may better suit execution requirements.
- Reduce overhead and degradation of rollback-recovery protocols protection and recovery tasks with the analysis of FT effects in the application execution.

This thesis focuses on providing a multi-platform fault tolerance architecture suitable for traditional bare-metal clusters, private and public clouds. This would represent the answer for the current demand of HPC resources, and its trends towards an ubiquitous, convenient, on-demand computing resources. At the same time, offer improvements over current FT protocols in terms of configuration. Finally, provide FT with a transparent, automatic, and distributed solution, which is configurable taking into account user, application, and runtime requirements.

1.1 Motivation

Fault Tolerance is required to provide high availability to MPI parallel applications. Most of parallel distributed applications rely on the MPI standard, which has several implementations such as: OpenMPI [1] and MPICH [2]. As early mentioned, they follow by default a *fail-stop* semantic, which means when a permanent failure occurs in a cluster's node, makes the execution aborts, requiring the re-execution of the application. This affects the total execution time and produces performance degradation affecting resources usage, incrementing energy consumption and users time expectations [11, 17, 35].

As the HPC systems demands is continuously growing the failure probabilities are also increasing. Cloud systems are not an exception for this phenomenon. Even with the replication and high availability technologies are offered by public and private clouds, they are aimed on transactional application. Meanwhile, parallel computing performed with MPI parallel programs, are stateful applications, requiring the FT layer of protection.

FT implementation can substantially change according to the execution environment. Traditionally, HPC applications executing on bare-metal hosts rely on FT solutions that are designed to operate on a reduced and static number of physical resources. That is, in case of host failure, traditional FT often attempts to continue execution on remaining

[1] https://www.open-mpi.org
[2] https://www.mpich.org

available working hosts. This approach allows to preserve the execution state and applications to finish without disruption. However, the performance of the remaining execution can be significantly degraded after a fault. Meanwhile, cloud ecosystems, presents an unique opportunity to eliminate this limitation given the seamless infinite resources available and its ability to create and destroy virtual machines in seconds. Still, traditional FT mechanisms need to be revamped to fully leverage such cloud capabilities to remove the FT overhead out of the execution hosts while maintaining performance of the applications even in presence of faults. Furthermore, another benefit is that the cost can be optimized by selecting among different alternatives to store the FT protection (e.g. checkpoints) as for example block or object storage, virtual machines, distributed databases or parallel file systems, depending on the user's requirements. One of the major advantages of implementing FT solutions in cloud environments are the support for dynamic allocation of resources as well as the ability to recreate and scale virtual instances.

In order to deeply analyze the components of a FT implementation, the availability (A) of a system is defined by the Equation 1.1. It is possible to observe, one of the main components to minimize, in order to increment availability, is Mean Time To Repair (MTTR).

$$A = \frac{MTBF}{MTBF + MTTR} \tag{1.1}$$

The Rollback-Recovery protocols are widely used to provide FT and overcome failures [30]. The protocols are based on the simple idea of saving the state of the application, and whenever a failure appear, use the stored state to restore the execution. Most of FT solutions that follows Rollback-Recovery protocols usually implement at least some of the following defined tasks (Fig. 1.1):

- **Protection**: This task is in charge of storing to a stable storage the current state of the parallel application. This procedure is one of the main sources of FT overhead.
- **Failure Monitoring**: This task constantly monitors the execution of the parallel application processes along the computation nodes.
- **Detection**: When a failure is detected with the failure monitoring task, this task is in charge of triggering the recovery procedure.

- **Recovery**: After a failure occurs, this task begin retrieving the last protection data of the failed processes. Usually, the state is fetched from a stable storage, and it is used to restore the failed processes to continue the application execution.
- **Reconfiguration**: This task configures the execution environment after the failure. It uses the available resources to map the failed processes into them. One common situation is to have spare node available, so the reconfiguration sets the execution environment to use it. However, if a spare is not available, the reconfiguration usually overloads the current execution environment. As it is possible to notice, the execution environment may change causing performance degradation.
- **Fault Masking**: When a failure occurs, affected processes normally are relaunched on a spare node or another active node. Remaining processes should communicate with them in a seamlessly way. This task provides the masking, rerouting messages whenever is necessary to the new location of failure affected processes.

Described tasks come with some configuration issues, the protection tasks (protection, failure monitoring and detection), cause overhead for the application during failure-free executions, due to storage of checkpoint and depending on the protection type, message-passing events may also be stored. The storage of the state information commonly interrupts the execution of the user's application. Consequently, the performance of the storage device is one of the factors that determines the failure-free

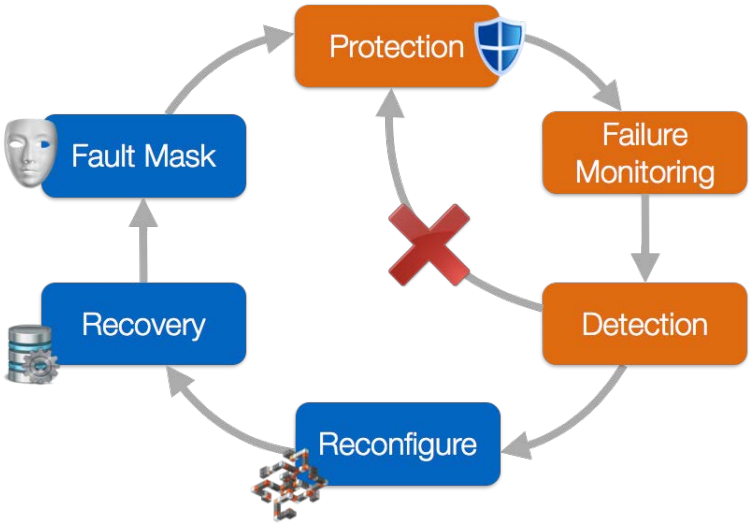


Fig. 1.1 Fault tolerance tasks.

overhead. Also, the recovery tasks (reconfiguration, recovery and fault masking), may degrade the system throughput depending on the resources that are assigned for FT tasks.

The protection, failure monitoring and detection are executed concurrently with the application. This means that it may affect the resources consumption, such as: cpu cores, main memory, storage and network. Basically, the FT implementation for HPC applications, protects the execution, though perturbing the user application resources.

One configuration challenge regarding the protection of the application processes, is the interval in which the checkpoints are created. Extreme shorter checkpoints intervals may forbid the application to continue execution due to the overhead of checkpoint creation during failure-free execution [34]. Meanwhile, taking checkpoints using a longer interval, let the application runs smoothly during failure-free execution, though when a failure occurs the recover can take significantly more, affecting the expected execution time of the user's application. Remove unuseful checkpoints can certainly reduce failure-free overhead.

It is important to define the invocation type of the checkpoints in the application. Time interval events can be used following optimal interval models, or event driven invocation can be done, allowing more control on what events can trigger checkpoints. Although, for this case, the application source modifications may be necessary, to insert checkpoints invocation.

Another configuration issue is regarding the resources that are used for FT protection purposes. Checkpoints and message-passing events can be stored on local to node disks or main memory, for performance purposes, though the application may not survive hard failures due to checkpoint information lost with node failures. A quite common configuration is to use centralized storage facilities. The storage resource for the protection is key definition, due to the significant amount of I/O generated and this may even block the progress of the application [8].

The degradation of the system is noticeable in presence of failures, when the recovery tasks are executed. If the FT solution is automatic, it does not depends on human intervention to restore and continue the application execution. Human intervention can be unpredictable, and the application can take an unpredictable time to be restored. An important matter, regarding the recovery is, how many resources in terms of computation nodes are addressed to overcome failures. When failures appear spare nodes can be used to resume the execution, although spare reservation may incur with an additional execution cost, that have to suite the users requirements.

Meanwhile, setting the application run without spare nodes, may result in the execution degradation after a failure appears.

This thesis is motivated by the challenges previously mentioned. It aims to the provision of an automatic, distributed, scalable and multi-platform FT architecture to provide resilience to parallel applications, which can be flexible and configurable to properly meet the users, applications, and runtime requirements.

1.2 Objectives

1.2.1 General Objective

The main objective of this thesis is to provide FT with a Multi-platform Resilience Manager (MRM) in the user and application layers, and at the same time reduces overhead and degradation. The FT configuration models, allows to protect the application execution taking into account the users, application and system requirements. The FT configuration provides system high availability to the application that can help it to finish, delivering to users the expected results.

1.2.2 Specific Objectives

- Provide a multi-platform resilience manager to support FT in the user-layer and application-layer for bare-metal clusters and clouds (public and private) using rollback-recovery protocols.
- Provide a cloud-native FT solution for HPC applications running on top of the message passing interface (MPI) with distributed and automatic solution.
- Provide fault tolerance configuration model to users.
- Provide flexibility and simplicity to users when configuring FT in their applications.
- Tune fault tolerance components for scientific parallel applications implemented with MPI, using user, application and runtime environment requirements.

1.3 Justification

Given the presented concepts about the fault tolerance topic, we propose to design a Multi-platform Resilience Manager (MRM), based on RADIC architecture, to provide

high availability to both: bare-metal clusters and clouds (public and private) with the rollback-recovery protocols.

We also aim to improve the configuration possibilities for protection and recovery tasks of the rollback-recovery protocols, by using models to setup checkpoint intervals without taking useless checkpoints. Moreover, the FT resources configuration is also tackled, by designing a model that allows to predict given a failure during an application execution, which is the configuration to resume the application. Moreover, it is able to determine if a spare is mandatory to continue the execution, and in case there are no enough resources to continue the execution, or the application is executing with a high degradation level, a safe stop is performed.

MRM focuses on automatic fault tolerance, which detects, recover and reconfigure the execution environment in order to reduce the MTTR. The manager offers two implementation flavors, the application and user levels, which allows both: FT user customization for their application, and FT transparency.

Specifically regarding the cloud environment, this thesis focuses on the challenge of integrating to cloud environments the transparency and distribution features of the RADIC architecture. The solution leverages the ability to dynamically allocate resources and make use of different storage offerings, as well as elasticity and performance diversity of available instances. The outcome is a resilience service for cloud environments.

1.4 Thesis Outline

Following the objectives defined to provide a FT multi-platform architecture for scientific parallel applications in order to provide high availability, this thesis is organized as follows:

Chapter 2: Thesis Background.

In this chapter a study of concepts used in this work for Fault Tolerance is done. Furthermore, state of the art solutions to tackled challenges are described.

Chapter 3: A Fault Tolerance Manager with Distributed Checkpoints for Automatic Recovery.

This chapter presents a Fault Tolerance Manager (FTM), which is part of the Multi-platform Resilience Manager. It distributes checkpoints along the computation nodes. It addresses the challenges to implement RADIC architecture for coordinated checkpoint protocol.

Chapter 4: Application-Layer FT with Dynamic Resource Controller.

This chapter presents the design modifications to the Multi-platform Resilience Manager in order to support an application-level FT solution. It leverage FTM combining the application-layer checkpoints with a message logger to implement the uncoordinated and semi-coordinated protocols. The solution also includes, a dynamic FT resources controller. Experimental design and evaluation is done using a Sender-Based Message Logger, and ULFM extension for OpenMPI.

Chapter 5: Configuring Fault Tolerance Protection and Recovery.

This chapter propose the models to configure the FT protection and recovery tasks. The configuration models are developed using coordinated and uncoordinated rollback-recovery protocols. Experimental design and results show the effectiveness of the proposed models.

Chapter 6: RaaS: Resilience as a Service for HPC in Cloud Environments.

This chapter presents a Fault Tolerance service for Cloud Environments. It is build on top of FTM and provides a multi-user FT service for cloud ecosystems.

Chapter 7: Conclusions and Open Lines.

This chapter summarize the work of this thesis, and state the open lines for future works.

Chapter 2

Thesis Background

"If I have seen further it is by standing on the shoulders of Giants."

Isaac Newton

This chapter presents background concepts, architectures and tools employed during the development of this thesis. First, the fault tolerance for HPC systems concepts are introduced. Then, the rollback-recovery protocols are further described. We also present the multiple implementation levels of FT. The RADIC architecture, which is used as baseline for this thesis is deeply revised. Finally, state of the art related to FT implementations in the application and user levels for cluster and cloud environments are analyzed to obtain relevant concepts to elaborate the contributions of this thesis.

2.1 Fault tolerance in HPC

High-Performance Computing using cluster and cloud systems is in the roadmap for progress in many scientific and engineering areas, including countries national security. Higher failure rates are expected with the growing scale of HPC systems. They will require higher complexity hardware (heterogeneous cores, deeper memory hierarchies, complex topologies) and software for these architectures will also become more complex, hence more error-prone [66].

Both, hardware and software future, will require more sophisticated energy and failure management. Moreover, todays energy efficient requirements, are making operation frequency diminution for processors, which allows the reduction of the voltage operation in order to obtain lower consumption rates, though increasing the failures probabilities [69].

Parallel distributed applications are build using a message passing interface (MPI), implementation. MPI follows by default a *fail-stop* semantic that aborts the execution, in case of failures. There are two main types of errors that threaten the computations of HPC applications implemented with MPI [36] executing in a cluster:

- **Soft-errors:** some of this errors are undetected data corruption errors. They are also known as Silent Data Corruption (SDC) or silent errors. When this kind of error appear, the user's application seems to run smoothly, but upon it finished its execution, bad results are produced [31].
- **Permanent-errors:** when a fault on a component lead to failure and it cannot continue its operation until it is repaired or replaced.

The contributions presented in this thesis are focused on only permanent errors. Although, an use-case for soft-errors protection and recovery is built by using part of this thesis contribution Montezanti et al. [51].

In order to build efficient FT solutions for HPC, some of the following concepts have to be combined:

- **Avoidance:** refers to the reduction of errors occurrence. For example, some FT solutions evaluate several metrics in the execution environment, to predict and avoid failures [28, 37].
- **Detection:** for detecting errors as soon as possible after their occurrence. Sometimes, depending on the solution, there could be a detection lag that depends on the parallel message passing library implementation. For example, a failure detection can be done when a message sending operation has failed.
- **Containment:** for limiting the impact of errors. Following this strategy for contention purposes, MPI [36] specification suggests that implementations should trigger an abort message to all processes of an application execution.
- **Recovery:** for overcoming detected errors. It represents the task done after a failure is detected, in order to continue application execution.
- **Diagnosis:** for identifying the root cause of detected errors. Several sources can be the origin of failures, it is important to define the scope of the FT solution.
- **Repair:** for repairing or replacing failed components. Several options to repair the failure affected resources can be used. Probably, the most traditional approach,

is the replacement of a failed computation node when failures appear. This is done to keep the system availability and allow the application finish its execution. Although, the repair can also aim to repair the application execution by using the remaining resources.

The concepts, previously presented, can be used to design FT solutions for multiple platforms.

2.1.1 Fault tolerance in bare-metal clusters

HPC execution systems are formed by several compute resources forming clusters for parallel and distributed processing. MPI library implementations, which became a *de facto* standard, are used to enable parallelism and communication among the multiple processes during an execution.

The main issue regarding failure probability is that clusters are often mostly formed by the addition of several computation nodes, and as it is already well-known the nodes are more failure-prone with the maturity and age of installation [78]. One example of it, is the Titan supercomputer in the Oak Ridge National Laboratory, which evolved from the Jaguar supercomputer installed early in 2005, and upgraded into Titan making it the No. 1 in the world, around 2012. Titan has at least 1 node fail-stop error per day, and around 350 ECC errors per minute [39, 40].

Fault tolerance is critical for long running parallel distributed applications executing in HPC systems. These applications requires a fault tolerance technique that should be independent of the cluster scalability. Fault tolerance becomes inevitable for MPI applications as unpredictable disruption can result in complete restart of the user's application. This makes the whole HPC system inactive and unproductive for a considerable amount of time, which increases the MTTR, and at the same time reduces the availability (Eq. 1.1). In order to avoid such loss, providing an automatic fault tolerance solution is required [23].

The resilience techniques have to be designed taking into the intrinsic redundancy nature of clusters, allowing users to continue their execution in remaining compute resources. Moreover, spare node support is necessary for FT solutions, to maintain the initial throughput of the application execution, in order to provide results within a limited period of time. Requesting spare nodes in a busy computing center may require a non-trivial amount of time and may be uneconomical. Although, In some cases, users may want to pay the penalty of a higher expected execution time avoiding the spare node costs.

Bare-metal clusters are mainly formed by similar compute nodes forming the mentioned intrinsic component redundancy. Although, the user's resources reservation for executions are rather static, meaning that during the execution, it is not simple to modify initial reservation. Moreover, the MPI environment built has to be also updated. FT solutions for bare-metal clusters, need to keep in mind this characteristic. Users require an automatic FT solution for their MPI parallel applications, and the support for configuration where resources may not be available, such as spare nodes. Furthermore, models to predict the impact of FT configurations for applications execution, can help users take decisions with insight information.

2.1.2 Fault tolerance in cloud

Cloud computing has dramatically changed the way web-scale operators provide services and manage infrastructures. The avoidance of in-house infrastructures and the acceleration of time to market are also fundamental aspects of this evolution. This disruption has been mainly driven by economical factors derived from the use of shared commercial off-the-shelf resources collocated in geographically distributed data centers. Today, however, cloud is evolving beyond cost benefits as providers are offering high-performance resource types including compute or memory-optimized instances, as well as GPU or FPGA options [3] and even bare-metal environments [55, 56]. Moreover, enhanced networking is also possible with single root I/O virtualization (SR-IOV), and tuned parallel file systems enable SSD-like storage speed [59]. The evolution in terms of performance, together with the multiplicity of pricing models, elasticity and high-availability make cloud a very competitive platform for scientific computing and HPC vertical markets in general [42].

Moving HPC applications to cloud is challenging, particularly in guaranteeing the completion of parallel, long-running, stateful applications. Large cloud environments are also prone to failures, putting at risk the user's application execution. In such environments, the bare-metal hosts are stressed during the provisioning of virtualized resources [76]; this increases the probability of failures as described in [64, 28]. Furthermore, virtualized resources are directly affected by failures on the underlying physical hosts as well as misconfiguration or maintenance policies specific to the cloud provider [81], e.g. Google's Compute Engine can arbitrarily migrate running instances for maintenance tasks, affecting those parallel application not designed to tolerate disruptions. Many parallel distributed applications build on top MPI, which follow by default *fail-stop* semantic that aborts the execution in case of host failure in a cluster. In this case, the application owner needs to restart the execution which affects the

wall clock time and, also, the cost since it requires to acquire computing resources for longer periods of time [11, 17].

Traditionally, HPC environments executing on bare-metal hosts rely on fault tolerance (FT) solutions that are designed to operate on a reduced and static number of physical resources. That is, in case of a host failure, traditional FT often attempts to continue the execution on the remaining available working hosts. This approach allows to preserve the execution state and applications to finish without disruption. However, the performance of the remaining execution can be significantly degraded after a fault.

For cloud environments, traditional FT solutions must be redesigned to leverage native cloud characteristics, such as the flexibility of virtual resources provision for both, protection and recovery tasks of FT. FT architectures for cloud also requires capabilities of FT provision for several users, executing multiple application on different virtual clusters.

2.2 Rollback-Recovery protocols

Fault tolerance ensures continuity in parallel applications execution. Among available techniques, the most known and used are the rollback-recovery protocols. This approach is based on state restoration of the affected processes due to failure, re-executing them from a checkpoint created at a particular point during the execution of program processes [32, 9, 45].

The rollback-recovery technique is based on processes state restoration in case of failures, where a failed process can be restarted from saved checkpoint data. Furthermore, rollback-recovery is used to recovery purposes after failures in parallel systems, because its advantages of recovering days-long executions and the lower implementation cost [30].

Main Rollback-Recovery protocols are coordinated, uncoordinated and semi-coordinated protocols [30][17]. These protocols and tools are further explained in the following subsections:

2.2.1 Coordinated protocol

The coordinated protocol orchestrate the creation of a global consistent state. This protocol simplifies recovery and it is not prone to domino effect, because all processes of the application always restart from the most recent checkpoint.

The blocking coordinated checkpoint operation starts with the blocking of all communications after the coordinator broadcast a message to start the snapshot [12, 20]. When a process receives this message, it stops its execution, flushes all the communication channels, takes a tentative checkpoint, and sends an acknowledgment message back to the coordinator. After the coordinator receives acknowledgments from all processes, it broadcasts a commit message that completes the global snapshot [32].

Figure 2.1, shows an example for coordinated checkpoints creation. The execution environment is composed of 2 nodes (N). Each process ($P_1 \dots P_N$) of the parallel application generates m messages, described as: $m_{(i,j,k)}$, where i = source process, j = destination process and k = is the send sequence number. The checkpointing operation is usually performed within a defined checkpoint interval (σ). Every process generates a checkpoint file ($CK_{x,y}$), where x is the process id and y is the checkpoint id. A global coordination is performed for the global snapshot creation. As the communication buffers are flushed before the checkpoint creation, no messages are sent or received during the checkpointing operation. The checkpoint files are stored into a stable storage, which is robust against some hardware failures and has to be accessible even when failures affects the computation nodes, in order to perform the recovery process.

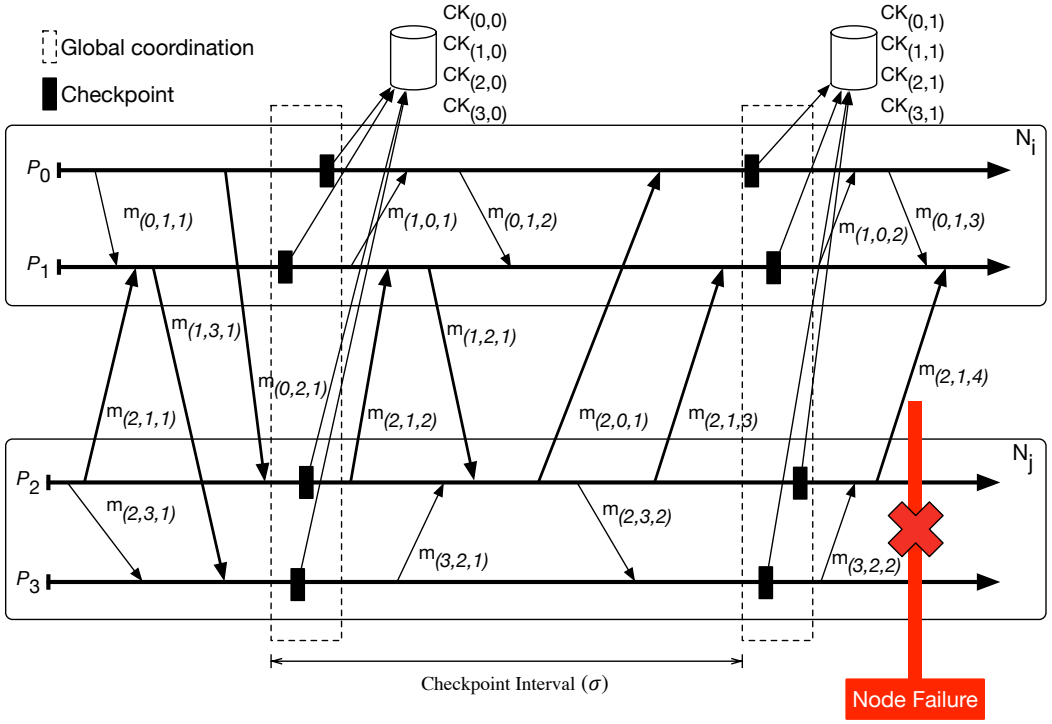


Fig. 2.1 Coordinated checkpoint protocol.

This approach can be suitable for a moderate number of processes, and the avoidance of message logger, which is used in semi-coordinated and uncoordinated protocols, can result in an energy efficient solution [25], although when the number of process increase the coordination can limit scalability and the concurrent access of a centralized I/O storage may degrade the performance. When failure occurs, all processes are forced to rollback to the most recent checkpoint even when only one process has failed and the execution continues from that point causing waste of computation in non-failed processes. An example is shown in Fig. 2.1, where node N_j is affected by a failure, which causes the whole application restart from the global checkpoint formed by the checkpoints $(CK_{(0,1)}; CK_{(1,1)}; CK_{(2,1)}; CK_{(3,1)})$.

In this work we use the coordinated protocol giving options according users, applications and runtime requirements.

2.2.2 Uncoordinated protocol

The uncoordinated checkpoint protocol combines process checkpoints with the storage of non-deterministic events, such as reception of messages, I/O operations; in order to restore the execution of the failed process and avoid the domino effect.

The protocol allows each process take checkpoint in the most convenient moment. Furthermore, in case of failures, only failed processes are restored, allowing remaining processes continue their execution. This is possible due to the checkpoint, plus the replay of non-deterministic events stored in an event logging facility. The events have to be replayed in the order they happened before failure to restore the processes [30, 32, 66].

Following this protocol, every application processes remain independent to each other. Although, if no message logging is performed, a recovery line has to be calculated during the recovery process after a failure appear. The recovery line may be difficult to obtain, because for its calculation an evaluation of non-failed process dependencies is necessary. After the recovery line calculation is done, if dependencies are found, it may require non-failed processes to rollback. The message logger avoids non-failed processes to rollback, by storing the exchanged messages. It warranties that the approach only re-executes computation lost of failed processes avoiding the waist of all-non failed processes computation. The logger also helps to avoid orphan processes. A process became orphan when it depends on a message, which is not sent back from another process which has not been rollback. [30, 32].

Regarding the moment in which the message logger stores messages to a stable storage, there are two main options *Pessimistic* and *Optimistic*. The *Pessimistic*

approach stores into the stable storage all messages before they reach their destination process. Meanwhile, in the *Optimistic* approach, messages are logged into a volatile storage, for faster delivery of messages to the destination processes. After the volatile storage, the messages are flushed into the stable storage [30].

Main message logging protocols are *Receiver*, *Sender* or *Hybrid* based [50]. The receiver based, usually stores messages in a stable storage after its reception; meanwhile, sender based stores messages in the computer node from which the message was sent. Hybrid-based uses temporally data structures in the sender and receiver performing a combination of sender and receiver based. Most of the available message loggers implementations are inside the MPI library, hence there is a need of a specific environment installation to run applications. Although, there are implementations in the application-layer. In [48] a message logger is designed in the application-layer to guarantee certain level of portability and independence to the MPI library.

Figure 2.2, shows an example of an parallel application using uncoordinated checkpoints. Each process $(P_1 \dots P_N)$ of the parallel application generates m messages, described as: $m_{(i,j,k)}$, where i = source process, j = destination process and k = is the send sequence number. Every process generates a checkpoint file $(CK_{x,y})$, where x is the process id and y is the checkpoint id. It is possible to observe that messages have to be stored. Two options are available, store them on the receiver or in the

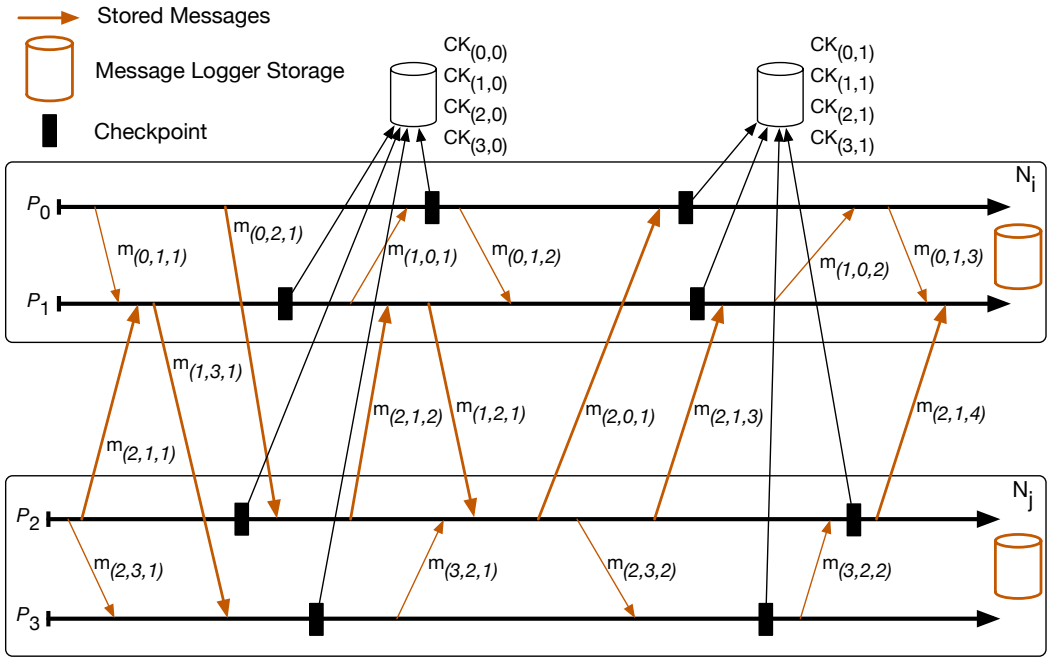


Fig. 2.2 Uncoordinated checkpoint protocol.

sender node. Considering a sender-based message logger and the process (P_2), the sent messages: $m_{(2,1,1)}; m_{(2,1,2)}; m_{(2,0,1)}; m_{(2,1,3)}; m_{(2,1,4)}$ are stored in the compute node (N_i). Garbage collection procedures usually identifies if a recovery line can be obtained from using the most recent checkpoint set, and in this case discard the stored messages [30].

In this thesis, regarding the message logging facility, we focus on an MPI library independent solution, so we combine the uncoordinated checkpoints protocol with a Sender-Based Message Logger in the Application-Layer [48]. It is important to acknowledge that the Sender-Based Message Logger less failure-free impact is expected in the applications, because messages are stored locally in the source node. Regarding the storage configuration, main memory is preferred, though, the memory consumption has to be monitored to avoid interfering with the application processes. FT resources consumption will depend on sending rate, communication pattern and size of the messages of the parallel application processes.

2.2.3 Semi-coordinated protocol

The semi-coordinated protocol combines the coordinated and uncoordinated protocols. This technique enables to create groups of coordination in which coordinated checkpoint are created. Messages exchanged among the coordination groups are logged. One of the most used coordination group criteria, is to group processes that are executing within a execution node [10, 17, 38].

The main idea with the semi-coordinated protocol is to reduce failure-free overhead by avoiding the storage of messages exchanged inside a computation node. Figure 2.3, shows an example of a parallel application protected using semi-coordinated protocol, where the coordination groups are defined by all processes executed in different nodes. Each process ($P_1 \dots P_N$) of the parallel application generates m messages, described as: $m_{(i,j,k)}$, where i = source process, j = destination process and k = is the send sequence number. Every process generates a checkpoint file $CK_{(x,y)}$, where x is the process id and y is the checkpoint id. Similar to uncoordinated protocol, messages are logged on receiver or sender node depending on the kind of message logger facility implemented. At the same time, coordination is done in the different coordination groups prior checkpointing operation.

In order to take advantage of multi-core systems, in this thesis we provide the semi-coordinated protocol using a solution which is in the application-level, offering yet another FT configuration option for users, avoiding high overhead due to message logging, offering a solution that is more scalable.

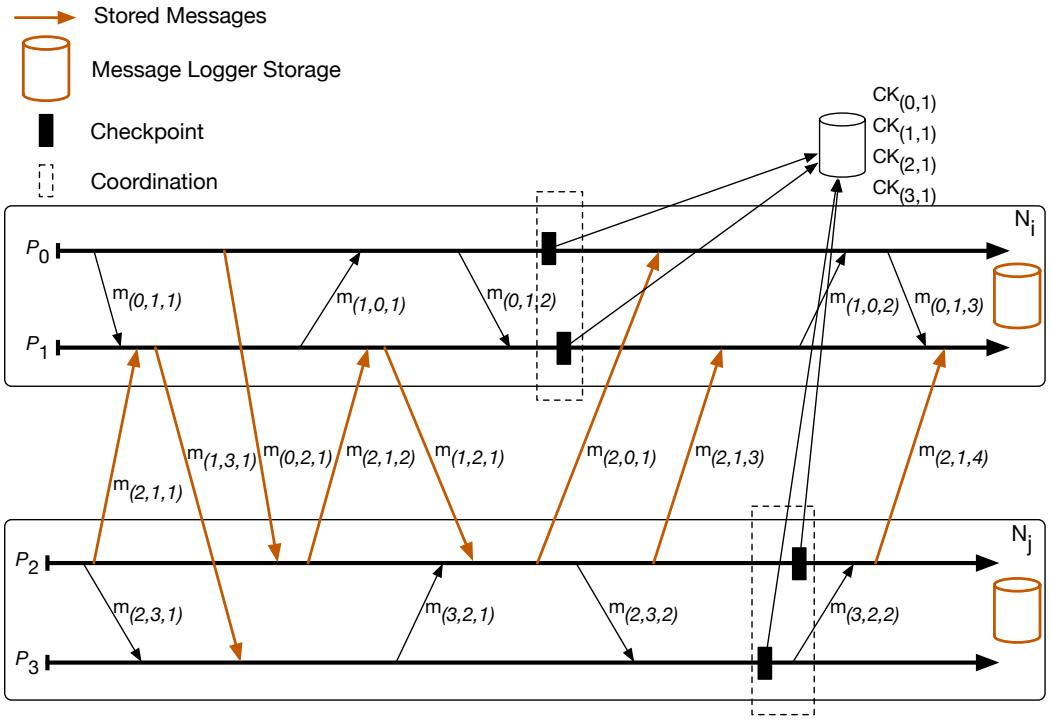


Fig. 2.3 Semi-coordinated checkpoint protocol.

2.3 Configuration of rollback-recovery protocols

This section describes checkpoint invocation methods and models aimed to configure and optimize rollback-recovery tasks for fault tolerance protection. This thesis focuses on how the characterization of the applications and execution environment can provide insights to generate suitable and specific rollback-recovery configurations for users.

2.3.1 Checkpoint invocation and interval models

Regarding the checkpoint invocation, there are two most known: *event driven* and *time* driven. For the event-driven checkpoint invocation, snapshots are initialized explicitly from the application, generally by using APIs provided by the checkpoint tool. Meanwhile, time-driven checkpoints, an interval of time must be specified to take checkpoints during the parallel execution. Depending on the knowledge of the application and the availability of the application source, one, both, or even a combination of both, approaches can be more suitable. For an application with no sources available, the best shot is to go with the time-driven, using a model to define the optimal interval. A FT manager with support of at least both approaches is desirable to provide higher levels of flexibility to the users.

To enable FT protection with rollback-recovery using for checkpoint time-driven invocation approach, it is necessary to define an interval in which checkpoints are taken. More than 40 years research have been done to find out the optimal checkpoint interval for rollback-recovery protocols [24, 22]. Daly [22] presented a deep analytical study for checkpoint intervals. Fialho et al. [34], proposed an interval model, which is designed specifically for parallel application, taking into account processes communications interactions to determine checkpoint intervals for parallel applications.

Depending on the moment in which a failure appears during the execution of a parallel application, a complete restart without FT protection may perform faster than restarting from a checkpoint took during the execution of the application with FT, in terms of wall time for users. In this thesis, we propose to characterize FT rollback-recovery protocols and determine main factors, which are used to design a protection model. The model is used with a methodology that helps to reduce the FT overhead. It uses the existing FT techniques, and intervals models. We propose a model that determines the starting point where FT becomes efficient using rollback-recovery protocols, for a parallel application running with FT in a particular execution environment. It helps users determine where they can start to really apply the checkpoints in the interval that was calculated using an interval checkpoint model, and in this way reducing the overhead by suppressing not necessary checkpoints.

2.3.2 Rollback-recovery storage configuration

Another major challenge when using the rollback-recovery protocols, is the storage configuration. One of the mayor disadvantages of the checkpoint techniques relies on the significant amount of I/O generated and this may even block the progress of the application [8]. A common storage configuration is to save checkpoints in a centralized stable storage, although there is usually a limited bandwidth due to the concurrency and it have to be shared along the processes of the parallel job [38]. Expósito et al. [33] present an analysis of I/O performance made on Amazon EC2 cloud environment, concluding that local disks (Ephemeral) are more efficient than a central storage such as EBS (Elastic Block Size).

To enhance availability, a very straight-forward and with decades of use approach is to perform replication. The replication of checkpoint files is a widely used method to provide high availability. Cunningham et al. [21] present how a resilient storage is implemented to survive node failure using a replication technique on X10 language. Moody et al. [52] designed a checkpointing solution called Scalable Checkpoint/Restart

(SCR) library for Application-Level checkpoints, which applies redundancy with three different levels: locally, in a partner, or using a XOR approach.

The multi-platform resilience manager is proposed in this thesis, which tries to take advantage of the node local storage bandwidth (disks and/or memory) to save checkpoints and distributes copies replicating checkpoints along the compute nodes of the execution environment, avoiding – when *possible*, the usage of a centralized storage repository.

2.3.3 Spare resources configuration

Configuring the spare nodes for a long-running execution is an open challenge [61]. When a failure appears and affects a computation node, FT mechanisms, usually uses spare node resources to re-configure the execution environment and resume the application execution. For busy bare-metal HPC clusters, the availability of a spare node is unpredictable, not to mention the costs of demanding a repair or replacement node ASAP. Meanwhile, virtual clusters hosted in cloud environments, have the advantages of supporting the dynamic allocation of resources.

When a failure occurs and a spare node is not available, the FT mechanisms re-configures the execution environment and resume the execution using the remaining resources. Although, performance could be affected. The spare unavailability can substantially affect the Mean Time To Repair (MTTR) of the execution environment.

This work tackles different execution environment and applications requirements to enhance the FT recovery task. Specifically, a methodology is provided to determine if the applications can resume its execution without having a spare node in case of failures. Moreover, the spare node configuration model proposed in this thesis, can be used to calculate *on-the-fly* effects of continuing the execution with and without spare node, allowing to make decisions.

For cloud environment, the multi-platform resilience manager, is enable to interact with the cloud manager, in charge of resource provision in cloud environments, to on-demand create and setup nodes to re-configure the execution environment after failures.

2.4 Fault tolerance implementation levels and tools

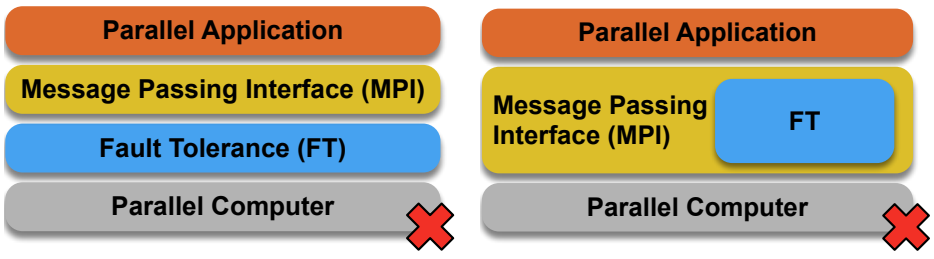
This section, explores the implementation levels in which the fault tolerance can be implemented to provide high availability for parallel executions [66, 16]. Additionally, state of the art well-known tools such as checkpointing facilities are described.

2.4.1 FT implementation levels

Several FT implementation levels can be used to provide fault tolerance. Each of them with its characteristics regarding transparency, customization and dependencies with the Operative System (OS). Most of the FT implementation level are the following:

- ***System-Level***: From the users perspective this implementation level is one of the straight-forward approach to implement FT in their applications, as no application modifications are required. In this implementation level two alternatives are contemplated to introduce FT (Fig. 2.4). FT can be suit at the operative system (Fig. 2.4a), or directly into the MPI library implementation (Fig. 2.4b). The main characteristic of this implementation level is the transparency, from the users and applications perspective. A drawback is the higher overhead it may incur, which is often due to the need to store additional process information, which is specific of the OS, including: process hierarchy, file and sockets descriptors ids, reserved and used memory including all process variables. Another inconvenient is the elevated costs of the software/hardware maintenance (solution life-cycle), as it requires constant updates to for e.g.: kernel modules, MPI libraries implementations, among others.

  Berkeley Lab Checkpoint/Restart (BLCR), is a system-level checkpoint/restart implementation for Linux clusters, in which most of HPC applications imple-



(a) Below MPI library. (b) Inside MPI library.

Fig. 2.4 System-Level fault tolerance implementations.

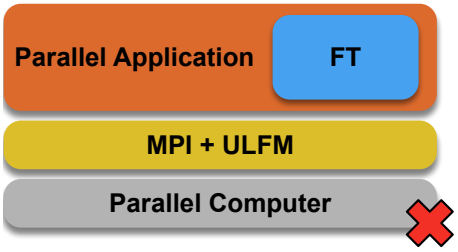


Fig. 2.5 Application-Level fault tolerance.

mented with MPI are executed. It works as a kernel module, to checkpoint processes [43].

- ***Application-Level***: This level of implementation allows users to define, specifically what data about the processes need to be stored, in order to keep the application executing in case of failures (Fig. 2.5). Some solutions, such as NR-MPI [68], FMI [62], FTI [5]; also provide interfaces, which allows users to specify the data structures which have to be saved during the checkpoint creation. The main advantage of this implementation kind is the customization of the protection, often obtaining lower overhead impact during the failure-free execution. On the other hand, applications source code, have include checkpoint invocation and to specify data to be stored. Also, the errors detection have to be implemented. Furthermore, the application source code must be modified to enable the restart reading the checkpoint files. Although, the application algorithm usually remains intact [2, 49, 6, 58].

  One of the most used tools to provide implementations of FT solution in the application-level is User-Level Failure Mitigation (ULFM). It was proposed by a working group in the MPI Forum to address a fault-aware MPI. Bland et al. [7], a study is developed showing that ULFM has almost no impact in terms of performance on a series of master-worker and highly coupled applications. ULFM main goal is allow failures mitigation on MPI application. It gives the execution control to the users when failures are detected and also delivers information regarding the failure type, and what processes has failed.

- ***User-Level***: user-level implementations perform the protection by linking the application with a checkpointing tool (Fig. 2.6). The solution acts in the user permission scope, hence, no administrator privileges are necessary to checkpoint the application. This implementations basically wraps system calls in order to access descriptors used by the application. Although, the checkpointing process

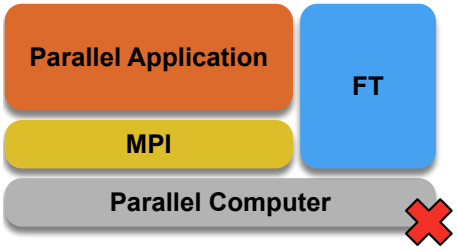


Fig. 2.6 User-Level fault tolerance.

is transparent, with some implementations, the application programs need to be modified so they include the tools in the compilation process [4, 57].

Tools that can be used to implement user-level FT solutions are checkpointing tools such as Esky, Condor, libckpt [30]. Ansel et al. [4], presented DMTCP (Distributed MultiThreaded CheckPointing), a checkpoint/restart tool for applications. It is one of the most known coordinated checkpoint facilities, which implements the coordinated checkpoint protocol, and does not requires administrative privileges to work. Furthermore, no modifications are needed to checkpoint applications with DMTCP.

After the taxonomy of FT implementations, and its tools, it is possible to state that in order to provide fault tolerance, a combination of tools are required, as none of the cited utilities comes as a complete high availability solution.

In this thesis, we focus to provide fault tolerance in the application and user levels. To do so, tools such as DMTCP that allows to take checkpoints in the user-level, and ULFM, which can perform the detection of errors are necessary. We aimed to both implementation levels, in order to offer multiple options to the application users. By using an user-level solution, no source code modifications are necessary to protect the user's applications without considerable efforts. Meanwhile, application-level can give users the complete customization for the FT protection, allowing them to define protection information, making the protection more specific for the application. Furthermore, the protection of application-level solutions does not need to store OS specific information.

2.5 Fault tolerance solutions

This section elaborates on the available fault tolerance solutions in both application and user levels. We also, review current FT solutions for cloud environments. Lastly,

RADIC architecture, used as baseline for this thesis, is described along with its components and functionalities.

2.5.1 FT current solutions

It is natural that most of the FT solutions for parallel applications implemented using MPI, are designed keeping in mind the bare-metal cluster characteristics. The solutions are typically designed with the combination of several components, such as MPI library, checkpointing tool, logger facility, compression utilities, among others.

System-Level FT solutions

Sankaran et al. [60], proposed LAM/MPI, a checkpoint/restart framework, which is implemented in LAM, an MPI library implementation. The solution, uses BLCR to create a coordinated global snapshot of the application processes. LAM/MPI is not automatic, meaning the intervention of an administrator to restart an application when failures appear. OpenMPI provides the abilities to perform coordinated checkpoints and restart processes transparently, based on LAM/MPI, it also uses the BLCR checkpointing tool [44]. Although, since v1.6 series of OpenMPI, its support was removed.

MPICH-V1 [65] propose a FT solution using a receiver-based message logger. The logger works with Channel Memories (CMs) used to store sent MPI messages, and dedicated nodes store checkpoints. Meanwhile, MPICH-V2 offers improves the high latencies of MPICH-V1 by using a sender-based message logger. Both solutions stores the protection information in a centralized manner, creating potential bottleneck problems [11].

RADIC MPI, RADIC-OMPI, and RADIC-Socket are solutions that provides high availability to parallel applications using BLCR checkpoint facility [17, 35, 26]. RADIC MPI, was designed implementing a subset of operations defined by the MPI standard. Meanwhile, RADIC-OMPI required modifications to the OpenMPI library implementation, which by the time it was modified was following the MPI-2 standard specification. Currently, OpenMPI is fully compliant to MPI-3.1 standards standard. RADIC-Socket is provided as middleware and requires modifications to the cluster OS installation. The solutions offer error detection and they are automatic, not requiring human intervention. They follow uncoordinated and semi-coordinated rollback-recovery protocols, which are also suited with a pessimistic receiver-based message logger. Although, as the

implementations are in the System-Level, they offer a transparent FT solution to the application's users.

However, each presented solution has the advantage of not requiring modifications at all for the user's application source code, some of them still requires human intervention, as previously mentioned (LAM/MPI, OpenMPI+BLCR). Moreover, each solution has the maintenance costs issue, constantly requiring updates in the implementation due to new upgrades on MPI implementation, kernels, checkpointing library. Another more serious issue, is the fact that now-days most of them are outdated, without any updates, and more importantly lacking support, making the FT solution deprecated and hard to adopt.

This is a motivation to explore the Application and User levels of FT solutions.

Application and user level FT solutions

In Moody et al. [52], designed and implemented a Scalable Checkpoint/Restart library (SCR) that writes checkpoints to RAM, or local node disks; it can apply redundancy in a partner or using a XOR approach. Bautista-Gomez et al. in [5] improve the XOR calculation adding an expensive Topology-Aware ReedSalomon encoding.

Another work is by Sato et al. [62], where they present a Fault Tolerant Messaging Interface (FMI) which act as a middle-ware between the messaging passing library and the user application. Users have change the programming model in their application by using an *FMI_loop* directive, which is use for synchronization purposes, and enables the library to write, rolls back, and restart the application processes.

NR-MPI leverages semantics of FT-MPI, into a FT solution that is in the application level [68]. It provides an API interface to modify applications in order to support FT. The solution offers failure detection and notification, to enable the placement of source code for recovery purposes. The main drawback is that it have to define *ghost-like* processes that are ready in case of failure, adding costs, and more importantly, the number of added processes are not dynamic, limiting the tolerance for multiple failures.

Most of the application-level solutions relies on ULFM, an MPI extension which is an interface to provide semantics for process failure detection, communicator revocation and reconfiguration [49].

Most of the solutions aims to tackle part of the FT problem, though the ideas and tools can be useful to design an automatic fault tolerance.

Cloud environment FT solutions

Cloud systems are build on top of considerable large bare-metal clusters, which are not free of failures [76]. This affects the layers on top of cluster nodes [19, 77]. Although, several high availability solutions for cloud are already available, they are focused on transactional stateless applications. For parallel applications implemented with MPI, fault tolerance solutions are used to tackle this issue.

The FT solutions in cloud are mainly categorized as *proactive* and *reactive* [53, 19]. Proactive FT constantly monitors the system to make failures prediction. Meanwhile, reactive solutions, performs snapshots of the system, that are used during the recovery process. The main idea in proactive approaches is to make predictions in order to prevent the effects of the failures before they happen. Although, predictions may not be accurate, hence proactive FT solutions would not be appropriated when higher levels of availability are pursued. In this work, we aim for reactive FT solutions.

In this work, host and VM permanent failures are addressed following rollback-recovery protocols [30, 66, 44] implemented in a FT service, taking the advantages of cloud computing resources. Rollback-recovery protocols are based on state restoration of the affected processes due to failure, re-executing them from a checkpoint created at a particular point during the execution of program processes [32, 9, 45].

Several reactive FT solutions in cloud are currently available to offer high availability. Regarding transactional applications, in [47] a framework is proposed, designed in a system-level and modular perspective, to provide FT in clouds. It performs VM instances replication to protect the execution environment of a client, instead of protecting the applications themselves [46]. The proposed framework becomes a costly solution compared to only protecting application executions. An HAProxy architecture is proposed in [19]. It employs job migration and replication techniques to provide high availability. The architecture has one node which performs redundancy for other compute node. A proxy node acts when one of the nodes fail by keeping the computation operative redirecting the requests to the active service. Another, solution is presented by [80], which is called BFTCloud, offers a FT solution for bizantine failures, with a replication policy in which a request must be processes in different nodes. At the end of the computation, the results are compared. We focus on providing an FT solution as a service, in order to protect parallel stateful applications against permanent failures for several clients, using multiple virtual clusters.

In Gómez et al. [41], a multi-cloud FT framework is introduced, which aims to tackle two problems regarding cloud provisioning: cloud providers variability, and fault tolerance for bag-of-tasks applications. The proposal make use of an Elasticity

Engine, which detects, loss of application performance due to virtual machines (VM) failures, hence not achieving the target performance; and the cloud provider variability, increasing the number of VM when target performance is not maintained.

For parallel applications, Cao et al. [14] proposed a checkpointing service for cloud environments. It enables application checkpointing and performs migration on heterogeneous cloud environments. The solution uses a centralized approach for the storage of the checkpoint files. Egwutuoha et al. [27] present a proactive FT solution, which relies on monitoring the VMs health status to act in case a failure is predicted. It is composed of modules to live-migrate VMs, failure prediction and a controller [29].

2.5.2 RADIC Architecture

This section introduces the RADIC, which is a fault tolerance architecture for parallel applications. The thesis uses the conceptual elements of RADIC to contribute with a multi-platform (clouds and clusters), FT solution.

The RADIC architecture (*Redundant Array of Distributed Independent Controllers*) provides FT for message passing parallel applications. It consists of a fully distributed array of FT controllers. The architecture performs protection, detection, recovery and error masking functions to guarantee a complete execution despite failures on the computing nodes. The functional components that create the distributed fault tolerance controller are the *protectors* (T) and the *observers* (O). The components are illustrated in Fig. 2.7.

The architecture offers high availability in an automatic, decentralized, transparent, configurable and scalable way. The functional components that create the distributed

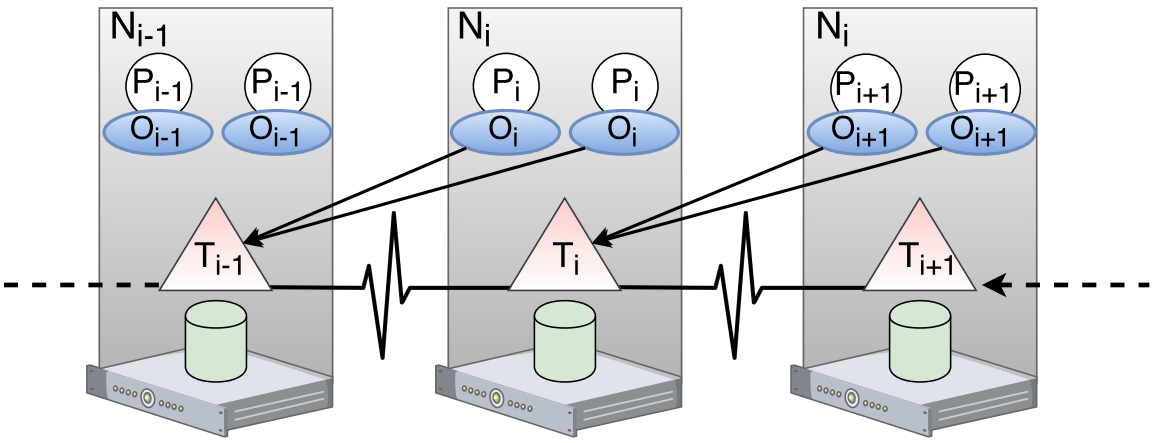


Fig. 2.7 RADIC Components: a Protector (T) for each Node (N) and an Observer (O) for each Application Process (P).

fault tolerance controller are the *protectors* and the *observers*. When a parallel job is launched both components are initialized along:

Observers For all processes (P) of the parallel application there is an observer attached. The observer intercepts every inbound and outbound communication of the process to a message logger when necessary, and also periodically stores checkpoint. Fault masking is done by this component during the recovery of the failed processes. Furthermore, this component is in charge of redirecting the messages to the new processes node locations after failures.

Protectors On each computer node (N) where the parallel application is running, one protector process is launched, which is in charge of the protection management of the processes in its neighbor node. Using *heartbeat/watchdog* protocol, protector neighbors constantly send and receive heartbeats from each other. In this way, it is possible to detect node failures within a desired period. The protectors receives checkpoints and logs from the observers running on the protected node.

The protection information is distributed along the computation nodes of the cluster, meaning a distributed FT approach. In case of failure, this information is used to restore the application execution. Spare nodes are used to replace failed nodes, although if there are no spare available, the execution may continue on the remaining nodes. This is a suitable approach for bare-metal clusters, where resources are limited and statically defined prior to execution.

2.6 Summary

As previously presented, the background concepts are helped researchers to design and elaborate current high availability solutions. These concepts represent the basis for the following chapters of this thesis.

Related works solutions are aimed to provide high availability for parallel applications in several, levels, such as user, application, or system. Each of these levels comes with benefits and drawbacks for the users to suit their applications with FT.

The background and related work are used in the following chapters of this thesis to design the Multi-platform Resilience Manager (MRM) in the user and application levels. This manager focuses on providing high availability using rollback-recovery protocols, for bare-metal clusters and clouds (public and private) taking into account users, applications, and execution environment requirements.

Chapter 3

Fault Tolerance Manager with Distributed Checkpoints for Automatic Recovery

"I have no special talent. I am only passionately curious."

Albert Einstein

3.1 Introduction

This chapter describes the Fault Tolerance Manager (FTM), which is in charge of FT protection for parallel applications using coordinated checkpoints. FTM allows the parallel creation and storage of checkpoints taking the advantage of high bandwidths in local stable storage (disk or memory), improving the checkpoint time and avoiding the bottleneck storing the checkpoints into a centralized stable storage. This is possible by distribution and replication procedures designed in the manager.

The manager distributes copies of the local checkpoint files replicating them among the computation nodes, always keeping track of the checkpoint files, allowing the automatic restart of the parallel job in case of hard node failures. FTM is transparent to the application, meaning no source-code modifications are required. It is capable of detecting failures, re-configuring the system using spare or remaining nodes to perform automatic recovery, minimizing the MTTR. This proposal is based in the automatic and distributed principles of the RADIC Architecture [17].

Taking RADIC architecture, as baseline, modifications and additions are done, to support coordinated checkpoints protocol using DMTCP checkpointing facility [4]. The components of FTM were integrated inside the Protectors of RADIC. Furthermore, this proposal maintains RADIC architecture main principles. This work represents a step forward to the user's applications, because they are no longer required to modify nor re-compile their applications with a specific MPI library implementation, for FT protection. The system administrators are also benefited, due to needless of installing any specific libraries in the clusters to support FT.

This manager is part of the thesis contribution as an alternative for the user's application, allowing them to suit their applications with FT with a minimal cost in terms of implementation, as the proposal is transparent, and automatic.

3.2 FTM Proposal Description

Providing high availability to the user's application executions comes with considerable overhead for failure-free executions, and most importantly an unknown MTTR if the repair requires human intervention in case of failures. If no FT is applied, when a failure appears, the whole application has to be re-launched. Lost work depends on the failure moment, the worst case is when a failure appears near the end of the application execution, requiring the re-execution of almost all the application. The best case would be a failure at the beginning of the application.

The FTM focuses on the management of protection information and failures in the execution environment, focusing on maintaining a limited MTTR value. The FT tasks that FTM has to perform are: protection, monitoring, detection, re-configuration and recovery. The re-configuration and recovery are only performed in case of failures.

The manager presented in this chapter runs along with the user's application. It extends RADIC architecture to support coordinated checkpoints using DMTCP as an use-case. DMTCP is a distributed multithreaded checkpointing tool for applications. It is one of the most known coordinated checkpoint facilities, which implements the coordinated checkpoint protocol, and does not requires administrative privileges to work. Furthermore, no modifications are needed to checkpoint applications with DMTCP [4].

A coordinated checkpoint facility such as DMTCP stores all application process checkpoints to files on a stable storage. When a failure occurs, it is possible to restore the parallel job execution using the last healthy set of checkpoint files. Regarding the location of checkpoint files, the usual configuration is to store them in a central stable

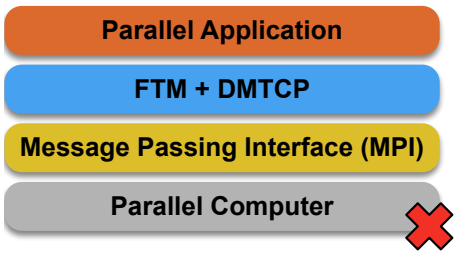


Fig. 3.1 FTM in a parallel system stack.

storage, warranting its access when a node failure appears. Although, centralized storage can become a potential bottleneck, due to the limited bandwidth that have to be shared with all concurrent processes access [38, 78]. For this matter, FTM adds a distribution and replication facility of checkpoint files along the computation nodes to avoid usage of a centralized storage, allowing the automatic restart after a node failure. It is important to remark that the proposal is designed taking into account also cloud environments, in which instance nodes are often offered with temporal storage capacities, and with this design we can avoid a centralized stable storage. The architecture stack is shown in Fig. 3.1 where the Parallel Computer is the component, which has failure probabilities. It is possible to observe that the proposal is on an independent layer, compatible with the application and user level FT.

FTM configures the checkpoint interval and the spare node usage for the FT according with: user, application and system requirements. The users can specify a limit regarding the execution time despite single or multiple failures. They can also limit the amount of resources assigned for the FT protection, such as memory, storage space and spare nodes quantity. By using the proposed models, it is also possible to evaluate several configuration alternatives, to help the FT configuration. Regarding the application, a checkpoint time (t_c) and size are needed, which are obtained by characterizing the application in the execution environment. System requirements, are the Mean Time Between Failures (MTBF) of the system, which is a statistical value, representing how often the system experience a failure, the amount of nodes used for computation and as spare nodes.

This proposed MRM manager, allows the parallel storage of checkpoints and takes advantage of node local stable storage (disk or main memory) high bandwidth improving the checkpoint time and avoiding the possible bottleneck of a centralized stable storage during checkpoint creation. When checkpoint files are stored locally (for e.g.: using DMTCP checkpointing tool) into the execution nodes, and a hard failure appears on one node, the access to their checkpoints files is lost, making restart not possible. FTM

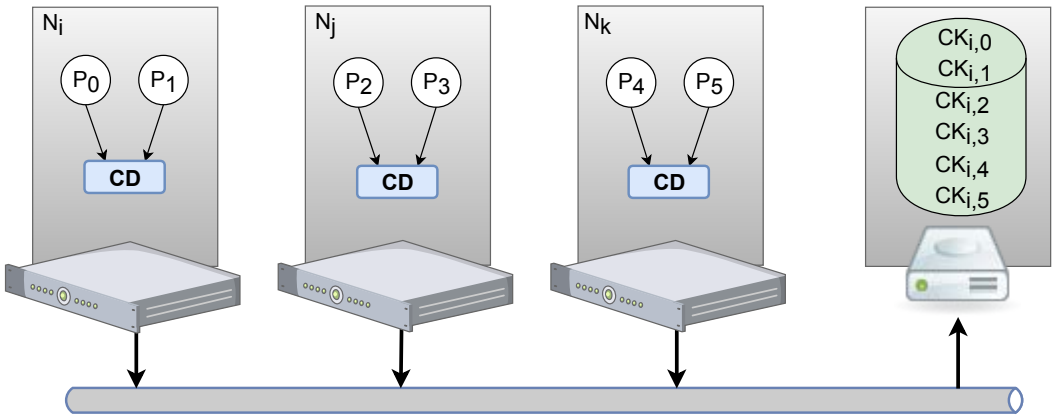


Fig. 3.2 Centralized checkpoint storage approach.

distributes copies of the local checkpoint files replicating them among the computation nodes, always keeping track of the checkpoint files, allowing the restart of the parallel job in case of node failures. The manager is transparent, as no modifications has to be made to the application source code. It also detects failures and re-configure the system using spare or remaining nodes to perform automatic recovery and in this way it minimizes the MTTR. This proposal is based in the automatic and distributed principles of RADIC [17].

3.2.1 Protection

This task is in charge of creating checkpoints for the processes of the parallel application. The checkpoint creation is a demanding task in terms of time and resources usage, especially when running a parallel application in HPC. Coordinated checkpoints implementations usually have a Checkpointing Daemon (CD) process running in each node of parallel execution environment. This daemon is in charge of the checkpointing procedure for the process running on the node. FTM configures the FT according with the user, application and system requirements. Although, a common configuration for checkpoint storage is to keep checkpoint files into a centralized stable storage, hence if a failure occurs, the files are reachable to restore the parallel execution (Fig. 3.2).

An alternative FT configuration would be to store checkpoint files into the local node storage. However, the potential issue of having the checkpoint files locally in the node is that when a node experiment a hard failure, the access to their checkpoint files is also lost. As the global checkpoint is the set of all checkpoint files for the application processes, then a part of the global checkpoint is lost avoiding the possibility to automatically restore the application after a failure appears. Although, with the usage

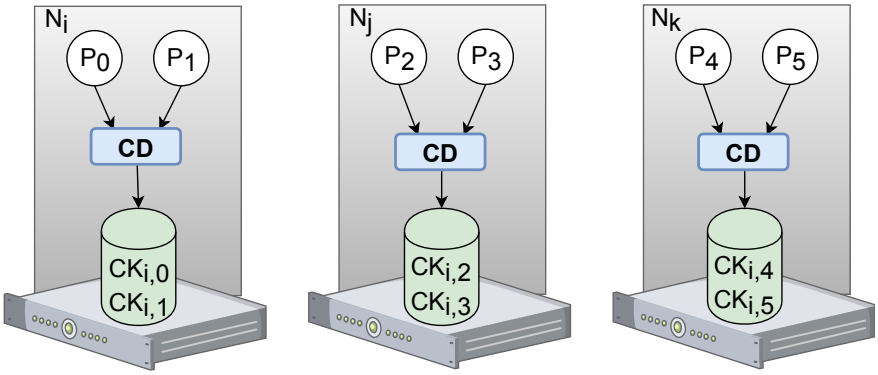


Fig. 3.3 Local to node checkpoint storage approach.

of multilevel checkpoint storage, older global checkpoints could be fetched and they can be used to restore the application, with the implication of losing more computational work. When FTM configures the local node storage for checkpoint files, it first have to determine if is possible to use local storage to save checkpoints, by evaluating the available local storage and the checkpoint files size. This approach is shown in Fig. 3.3, where the application processes are represented as $P_{(1...n)}$ running on a computer node $N_{(i...k)}$. In the local storage the checkpoint files are stored $CK_{(i...k,1...n)}$.

FTM is suited with a distribution and replication facility for checkpoint files, which is are used to maintain availability of the files even when a node is lost, and at the same time improving the checkpoint creation by taking advantage of the local storage bandwidth. For this matter, it replicates local checkpoint files to a logical neighbor node. The neighbor is selected taking into account that two neighbor can not share a common failure probable device, such as: network routers, switches or power supply. This configuration is a requirement to avoid the lost of neighbor nodes simultaneously, improving restart possibilities in case of failures. The Replicator (R) is inside the FTM manager and it performs the replication. It is integrated as a new component inside the RADIC's protector, which is attached to each node of the execution environment and contains the information of the neighbor node processes that it is protecting.

The replication process is illustrated in the Fig. 3.4. It starts when a global checkpoint is completely created. To trigger the replication, FTM watches the global checkpoint completion. In the particular case of DMTCP, an use-case scenario for this proposal, FTM is configured to watch for a file that is modified by DMTCP, when a global checkpoint is completely created (DMTCP Timing file). The replication is performed in parallel with the application execution, meaning that it does not blocks the application execution. This behavior is depicted in Fig. 3.5. The replication process may generate a lot of traffic in the network during the replication process,

though this issue may be addressed by using an alternative network interface for the replication. The checkpoint files are duplicated, although when new checkpoints are created, the checkpoint files and its replicated copies are deleted and replaced for new checkpoint files using a garbage collection procedure. For FT protection against soft failures, the older copies may be preserved and sent to a centralized storage in order to restore the application state [51].

For checkpoint interval configuration, FTM uses the well-known Daly's [22] model, which uses the MTBF, a statistical value of time in which a failure affects the execution environment [15, 63], and the application checkpoint time. The calculated checkpoint interval is then, configured to the DMTCP checkpointing tool. The application checkpoint time is measured considering the total time that it takes to coordinate the application processes, flush network communications, and store the checkpoint files into local storage. The replication time is not considered, as it is not in the critical path of the application execution. It is executed in parallel with the application execution. Although, the replication process consumes storage resources, as it creates a second copy of the checkpoint files stored locally. The replication implemented in FTM, consumes more storage resources, though it obtains faster checkpoint creation performance and consequently reducing the overhead.

3.2.2 Monitoring and Detection

The monitoring and detection tasks are implemented inside the FTM using the RADIC protector and observers components concept. The fault detection is required to perform automatic recovery. Despite MPI is fail-stop, a failure timeout has to pass till the

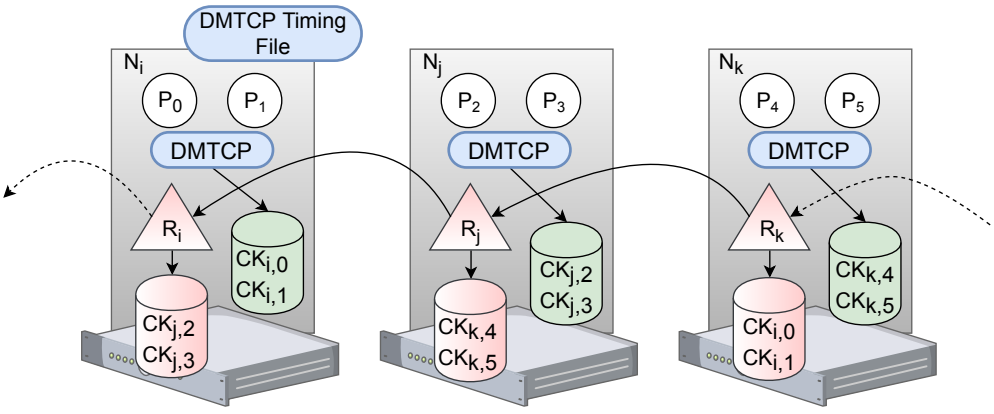


Fig. 3.4 Replication components in a FT configuration without spare nodes: DMTCP checkpointing facility storing checkpoints using the local storage. Replicators (R) copies checkpoints to their neighbors.

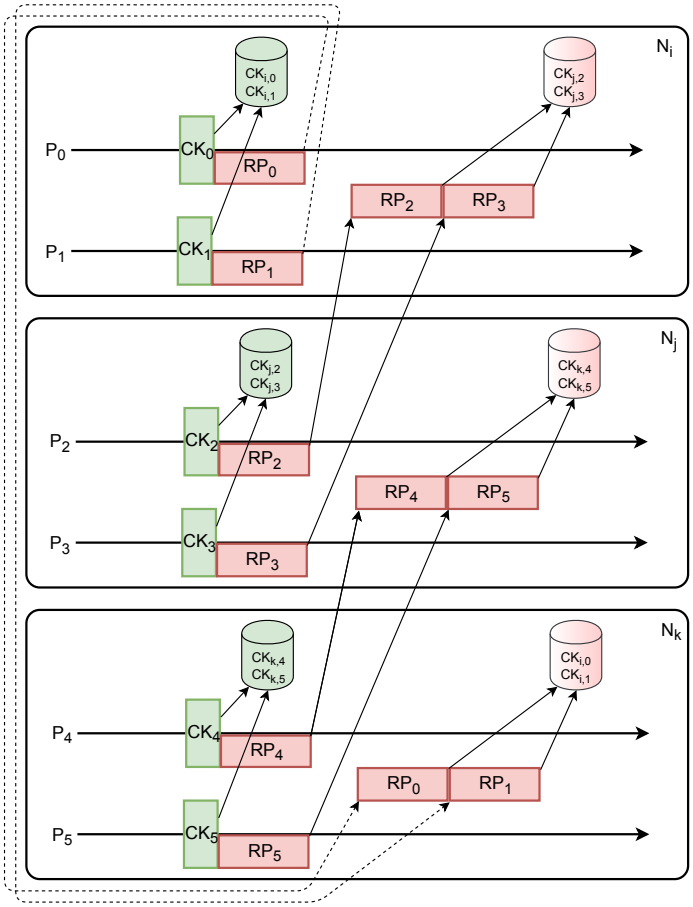


Fig. 3.5 Replication scheme: *RP* figures replicates checkpoint files to the neighbor node. The replication process runs in parallel with the application processes.

whole application crashes. For this reason, a frequency of node monitoring is defined for the heartbeat/watchdog mechanism of FTM to detect node and MPI application processes failures.

The process task is illustrated in Algorithm 1. For each computation node an infinite loop is executed till the application ends, verifying if its corresponding neighbor node (*neighbor_node*) is running. The verification is done according to the frequency of node monitoring ($freq\_monitoring$). When a failure is detected in any computation node, the task returns the failed node information that is used in the re-configuration task. The heartbeat/watchdog mechanism confirms the failure status by sending a message, to avoid false positive failures.

Algorithm 1 Algorithm for monitoring and failure detection

Input: $neighbor\_node$, $freq\_monitoring$
Output: Failed node (FN)
1: **function** WAIT($watchdog_interval$)
2: Wait for a period of time $watchdog_interval$
3: **end function**
Function to verify if a process is running on a node
4: **function** VERIFY_RUNNING($node$)
5: Connect to $node$
6: **return** $node$ is running
7: **end function**
Main Loop Block
8: **while** $app\_run = true$ **do** ▷ User Application is running
9: WAIT($freq\_monitoring$)
10: $running$ = VERIFY_RUNNING($neighbor\_node$)
11: **if** ($running \neq true$) **then**
12: $FN = neighbor\_node$
13: **return** FN
14: **end if**
15: **end while**

3.2.3 Re-Configuration

After the node failed identification, the re-configuration is performed. This task takes into account if a spare node is available. In case spare nodes are available, the restart is setup to use them, and remove them from the spare nodes set. Otherwise, the restart is performed using the remaining resources. When restarting the application in a reduced environment, all application processes from the lost node are configured to restart together in the neighbor node. Although this may reduce performance of the application execution, it is done in order to maintain the application original mapping, avoiding expensive communications due to new process locations. Furthermore, the neighbor already has the checkpoints to perform the restart.

Each checkpoint created by DMTCP generates a restart script, which has to be invoked when restarting the application. FTM automatically finds the restart script and modifies it to adequate the environment prior restart.

Further failures are supported, as the monitoring and detection tasks is re-started, using the list of alive nodes from the new execution environment. Although, if a failure occurs during the recovery process, the FTM will not recover the application execution. The re-configuration and recovery processes are considered as atomic operations of the FTM.

3.2.4 Recovery and Restoration

The recovery is performed using the environment set up during the re-configuration task. FTM always keeps track of the checkpoint files and its replicas along the computation nodes to make possible the automatic restart in case of failures. It uses a data

Table 3.1 Data structure for replicator process and spare node information for 4 compute nodes and 1 spare node.

Node	Neighbor	Node function
$node_i$	$node_j$	compute
$node_j$	$node_k$	compute
$node_k$	$node_l$	compute
$node_l$	$node_i$	compute
$node_m$	n/a	spare

structure similar to the RADIC table [50], without the messages control information, as coordinated checkpoints are used (Table 3.1), in which 4 computation node and 1 spare node example is observed. A repository path is also defined, e.g.: */tmp/rpfiles*.

To re-launch the application, FTM uses this information and restart the application. If a spare node should be used, FTM transfers the checkpoint files of the failed node to the spare node prior the restart. However, if the restart is done without a spare node, the lost node neighbor already has the checkpoint files to restart the application. The restart is performed using DMTCP, which re-creates all the application processes, load processes memory, re-bind sockets connection and fill the communication buffers. When using the remaining resources, the application may lost performance due to the overloaded node in which the application processes of the lost node are running. The lost work is performed till the failure point is reached.

To illustrate the recovery process using FTM, an application is running, and during its execution is taking checkpoints. FTM performs replication of the checkpoint files on the neighbor nodes. When a failure is detected, all processes and checkpoints of the lost node are lost (Fig. 3.6).

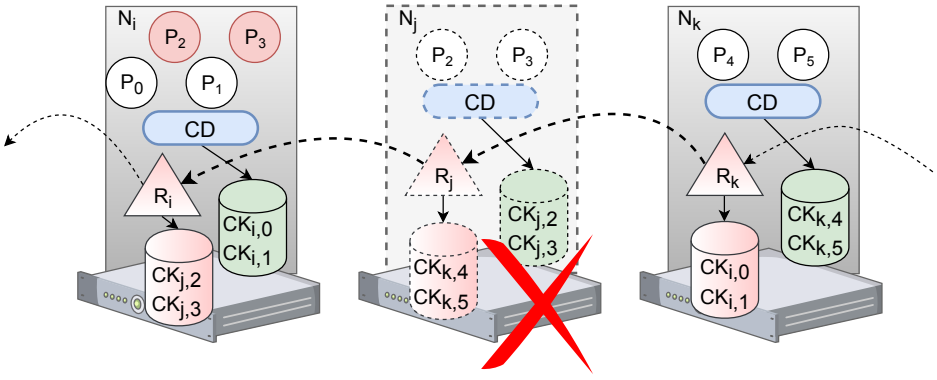


Fig. 3.6 FTM: Failure detection in a node.

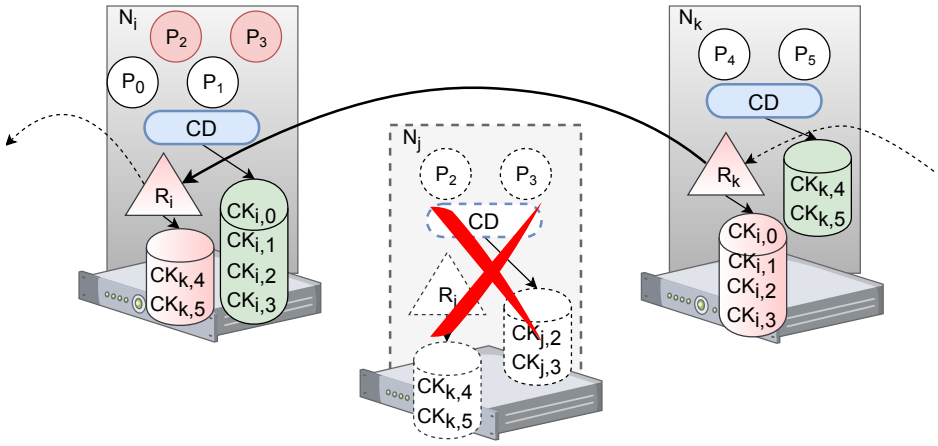


Fig. 3.7 FTM: After restart process.

The re-configuration is performed, taking into account that no spare is available for this scenario. FTM restores the execution of the parallel application using the remaining nodes (Fig. 3.7). The re-configuration also includes the setup of the checkpoint interval into DMTCP. The recovery process restores the execution and protects the application with the checkpoint interval configured during the re-configuration phase.

FTM reduces MTTR, performing automatic restart in case of failures, compared to manual restarts. It supports multiple failures during the application execution, though simultaneous failures in neighbor nodes are not supported, neither during a recovery process. It also takes advantage of local storage to accelerate checkpoint creation time. Using the concepts previously described, a MTTR using FTM can be defined as the Eq. 3.1.

$$MTTR = D_{time} + CF_{time} + MS_{time} + CR_{time} + RX_{time} \tag{3.1}$$

Where:

- D_{time}: latency time for failure detection.
- CF_{time}: time taken in the re-configuration process. The re-configuration uses the remaining nodes available, but it can also use a spare node to replace the failed one.
- MS_{time}: time in which the checkpoint is transferred to the spare node (if the spare node is available).
- CR_{time}: time in which the checkpoint is restored. Usually this value is approximately the same as the checkpoint time.

- RX_{time}: this value measures the re-execution of the application until it reaches the previous failure point.

3.3 Experimental Results

In this section a series of experiments are presented, which verifies the FTM functionality. Experimental executions are performed in 2 different clusters; a physical private cluster, and a virtual cluster built in Amazon EC2. This experiments were developed in a controlled environment and failures are injected manually to the computation nodes. Executions are performed at least 3 to 5 times, unless it is explicitly specified differently, and measurements are taken using the ***time***, system tool wall time value. Checkpoint and restart operations time measurements are obtained from the specialized plug-in built in DMTCP tool.

First, the overhead and cost evaluation of the FTM is shown. Then, the distribution and replication operations are analyzed, measuring the effects of different FT configurations, and how they impact on the application throughput. Finally, a complete scenario execution with FTM is shown including its FT tasks in order to perform automatic recovery from failures.

3.3.1 Hardware Configuration

The standard private cluster (AOCLSB), is composed of 8 nodes. Each node has 2 network interfaces one is exclusively to management tasks and other to perform computation.

Table 3.2 Experimental environment.

COMPONENT	AOCLSB	CLOUD
CPU	2 quad-core Intel(R) Xeon(R) E5430 @ 2.66 GHz	8 (vCPU) Intel Xeon E5-2680v2, 2.8 GHz
Memory	16 GB RAM	16 GB RAM
Local Storage	30 GB HDD	2 x 80GB SSD
Centralized Storage	NFS(v3): 715 GB	NFS(v4): 30 GB EBS
Network	Dual Broadcom NetXtreme IITM 5708 Gigabit Ethernet	High Performance
Nodes	Dell PowerEdge M600	c3.2xlarge

Meanwhile, for the cloud environment (CLOUD), 8 instances were rented from Amazon EC2. The launch and configuration of the instances were done using StarCluster [67]. Each instance is a HVM (Hardware-assisted Virtual Machine), this is a virtualization type, which provides the ability to run an operating system directly on top of a virtual machine without any modification. All the instances are rented from US East (N. Virginia) data center of Amazon. Both cluster information are shown in Table 3.2, and further details are described in Appendix A.0.1.

3.3.2 Software Configuration and Tools

Parallel applications for the experiments in this work were compiled using Open MPI 1.6.5. The DMTCP coordinated checkpoint facility used is the 2.4.0 version and timing measure option was configured. DMTCP has several configuration options and plug-ins in order to checkpoint applications. Although, we are focuses on the event-driven and time-driven options. For the event-driven users introduce checkpoint invocations directly in their application source code. Meanwhile, for the time-driven approach, a checkpoint interval time is required. As previously mentioned, the Daly's interval model ([22]) is used $\sigma = \sqrt{2\alpha \cdot t_c} - t_c$. Where, the checkpoint time is t_c and α is the Mean Time To Interrupt (MTTI) for a given system. For the experiments, 1000 seconds is considered as a default value.

3.3.3 Applications

For the experimental results, three applications were selected. Each of them uses different implementation paradigm such as: Master/Worker, Circular Pipeline and SPMD:

- **Matrix Multiplication**: A MPI based application, which implements a Master/Worker model. Performs a Matrix Multiplication of 2000 x 2000 elements.
- **N-Body**: This is a dynamic particle simulation implemented as a circular pipeline. The simulation is configured to run with 150000 particles and 10 iterations.
- **NAS-CG**: This is the Conjugate Gradient method implementation included in the NAS benchmark suite [54]. The experiments use the Class B, which have 75000 rows and 75 iterations. This application follows a SPMD paradigm.

3.3.4 FTM Overhead Cost Evaluation

In this section an evaluation of the overhead and cost produced by FTM is done. Comparisons are developed between applications executions with and without FT and with FTM. For the evaluation, 8 nodes of the cluster AOCLSB are used. Each application runs with 64 processes distributed equally in each node, hence every execution runs 8 processes per node.

Execution scenarios were designed to evaluate the cost in terms of overhead generated by the FT protection, and the proposed technique with two network interface configuration. The experimental scenarios are the following:

- **No-FT:** stands for executions without protecting the applications with FT.
- **FT:** executions are protected with FT, the checkpoints are stored in the nodes, locally without FTM.
- **FTM-CI:** executions are protected with FT, with FTM, distributing and replicating the local checkpoint files. The FTM is setup to use the compute network interface (CI).
- **FTM-AI:** executions are protected with FT, with FTM, distributing and replicating the local checkpoint files. The FTM is setup to use an alternative network interface (AI).

For matrix multiplication and N-Body applications, the FT configuration is setup with event-driven checkpoints. N-Body was configured to invoke a checkpoint, after every 3 iterations. With this setup, 3 checkpoints are taken during the application execution. Similarly, 3 checkpoints are also made in the matrix multiplication application. The checkpoint invocation are made in the following event: when the workload is partially distributed, completely distributed, and when the results are received in the master process from the workers. Meanwhile, NAS-CG benchmark was configured to take checkpoints using an interval of time. The resulting checkpoint interval time is $\sigma = 91$ seconds, using Daly's [22] interval model with the following values: $\alpha = 1000$ seconds and $t_c = 4.6$ seconds. The checkpoint time (t_c) is measured with an initial checkpoint. With this checkpoint interval, 3 checkpoint were taken in total during the NAS-CG executions. The storage resource usage along with the replication time are shown in Table 3.3.

The storage resource usage is duplicated due to the replication process in FTM. Only last checkpoint and the corresponding replicas are keep. Older checkpoints are

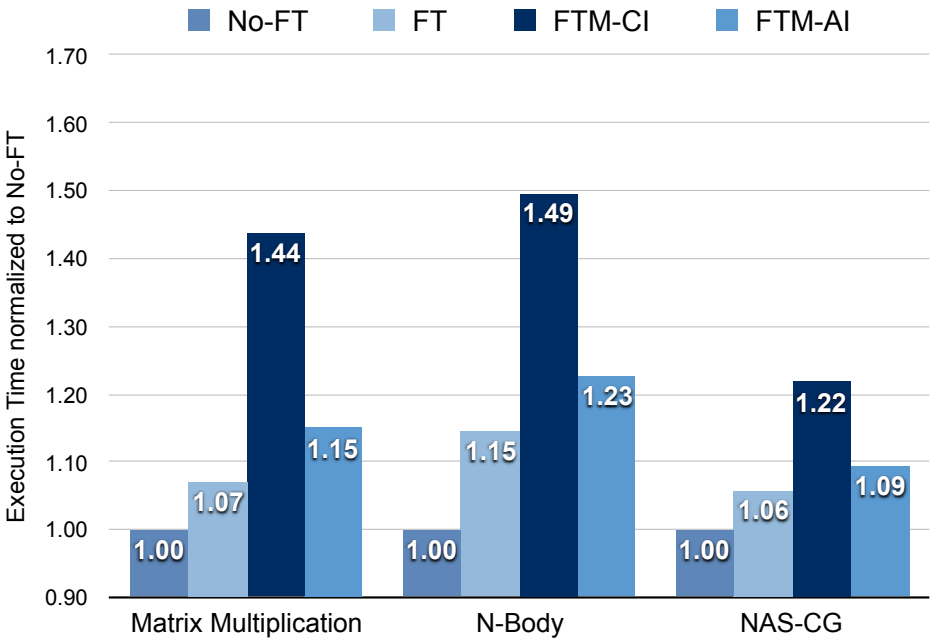


Fig. 3.8 Execution overhead.

deleted by the garbage collection. With this procedure we can calculate that minimum space required is to store 2 checkpoint plus its replication copies.

The resulting execution overhead costs are shown in Fig. 3.8. The executions time are normalized to the execution time without FT (No-FT). It is possible to observe that applying FTM protection costs between 9%-23% in terms of execution overhead. It is also remarkable how the network interface selection impacts on the application performance. By sharing the compute network interface the applications downgrade its performance up to 25% compared with using an alternative interface.

3.3.5 FTM Distribution Operation Evaluation

In this section the distribution operation in FTM is evaluated. Furthermore, the experiments results the effects of several FT protection configurations. Specifically,

Table 3.3 Total checkpoints size of 64 processes for each application in the AOCLSB cluster.

Application	FT (MB)	FTM (MB)
Matrix Multiplication	1128	2256
N-Body	1216	2432
NAS-CG Class B	1488	2976

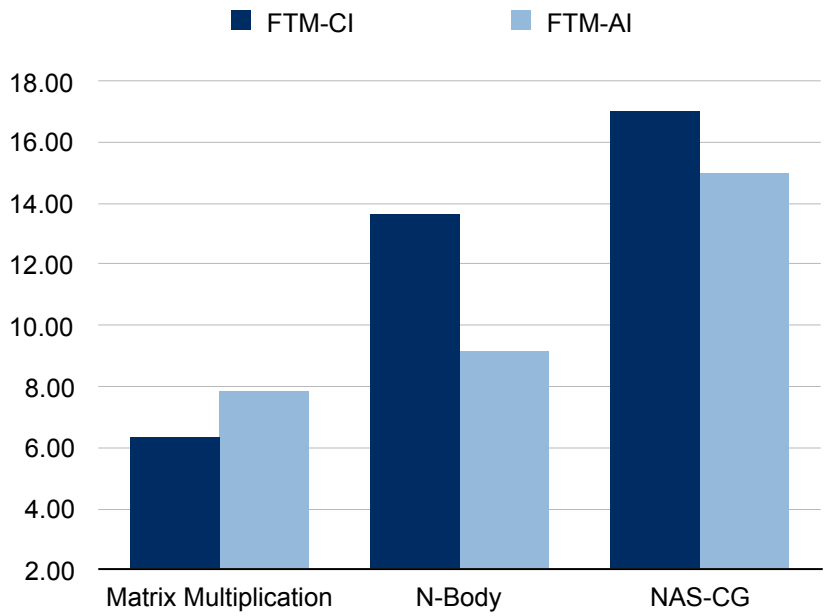


Fig. 3.9 Checkpoint replication using compute network interface (CI) and alternative network interface (AI) in the AOCLSB.

comparisons are made by configuring FTM protection to use a centralized storage approach vs local storage with the distribution operation. Also, the more significant variables of the MTTR are discussed.

To evaluate the creation and replication of checkpoints, two clusters were used, a private cluster AOCLSB and a virtual cluster built with the public cloud Amazon EC2 (CLOUD).

The distribution is evaluated by measuring the replication time of the checkpoint files, after their corresponding creation using the compute network interface (CI) and the alternative network interface (AI). The applications FT configurations are the same as Section 3.3.4. Results are shown in Fig. 3.9. It is possible to observe that using the same network interface as the application, influence the distribution operation. By using an alternative interface, the distribution operation does not interfere with the application obtaining improvements in application performance.

Centralized and distributed configuration approaches are evaluated using the Matrix Multiplication in both execution environments: CLOUD and AOCLSB. The CLOUD uses Ephemeral local disks for as local storage and mount a NFS filesystem using an EBS volume for the centralized approach. Meanwhile, the AOCLSB cluster also mounts a NFS filesystem using an external disk for the centralized configuration. Matrix multiplication application is configured to take checkpoints with the time-driven approach, with intervals of 60 seconds. Different quantity of checkpoints are

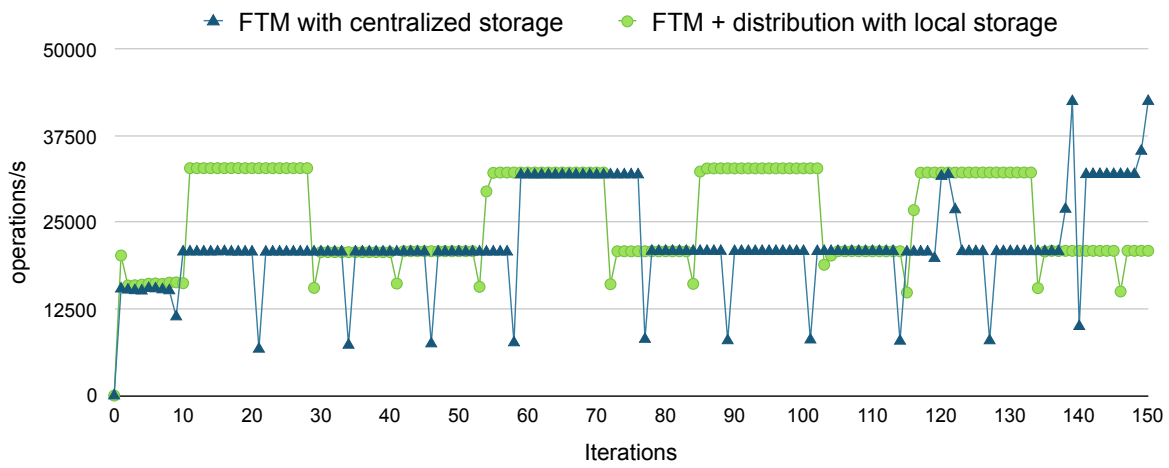


Fig. 3.10 Matrix multiplication using EBS and local storage on Amazon EC2.

expected, due to different execution performances obtained from AOCLSB clusters and the CLOUD.

For the executions, the throughput of one application process is measured on both execution environments: AOCLSB and CLOUD, in order to observe how different configurations are affecting the application execution. The throughput in CLOUD is depicted in Fig. 3.10 and in AOCLSB is shown in Fig. 3.11. Each fall in the application throughput represents a checkpoint creation.

The checkpoint distribution effect is noticeable in the throughput of the application running in both clusters, as a little drop after the checkpoint creation. Both environment shows that when using the centralized approach, a larger penalization on the application

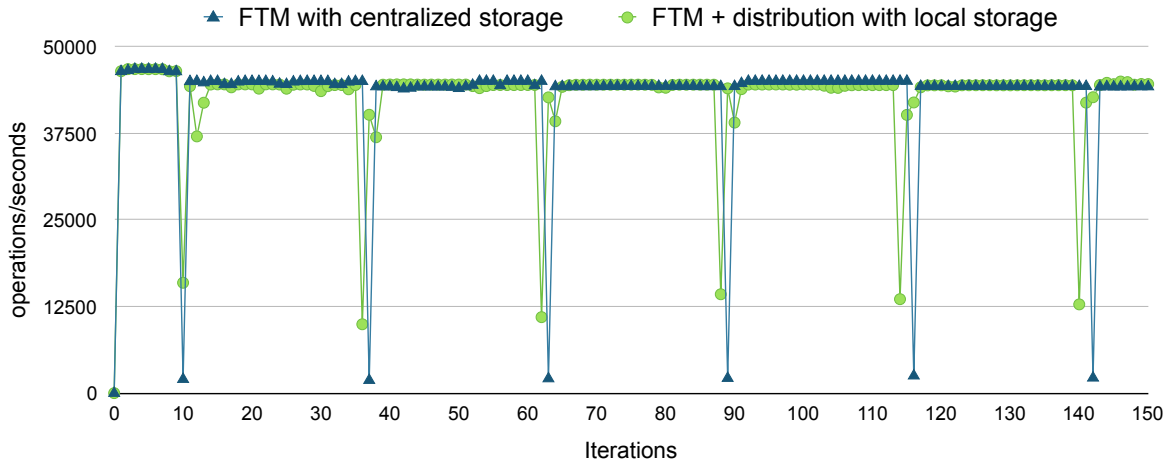


Fig. 3.11 Matrix multiplication on AOCLSB cluster using local and centralized storage NFS.

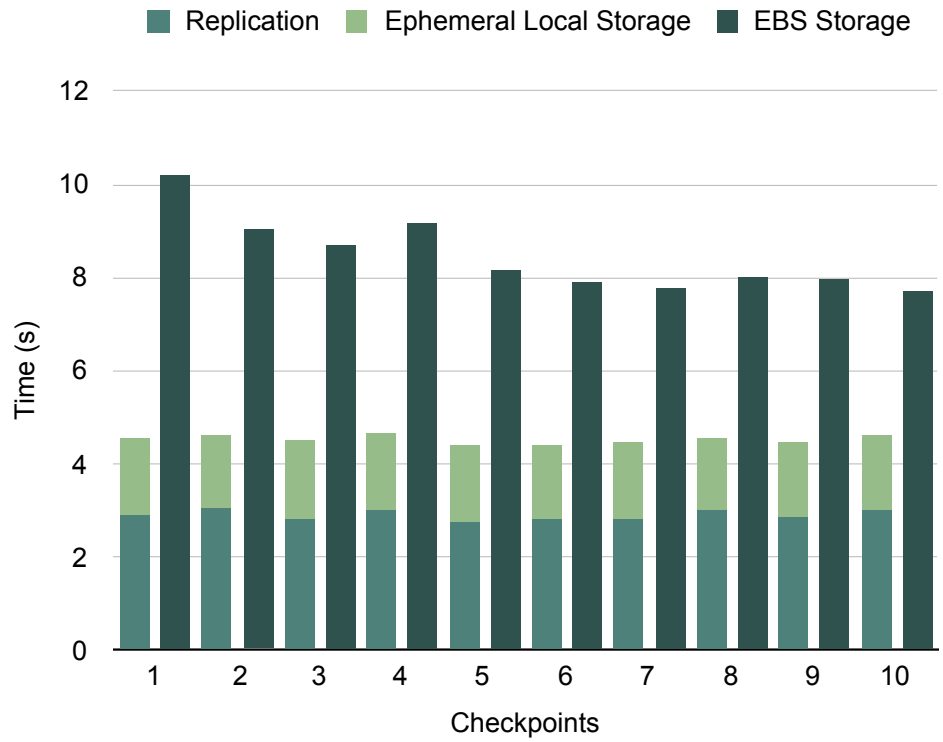


Fig. 3.12 Checkpoint creation time of a matrix multiplication execution in Amazon EC2 with c3.2xlarge instances.

throughput is obtained, compared to the usage of local storage and the FTM distribution approach. It is possible to observe that the CLOUD has more variability compared to the bare-metal AOCLSB cluster. The application throughput is more constant and bigger when executing in the AOCLSB cluster, in terms of operations per seconds, hence faster execution of the application is done, taking less checkpoints, as the checkpoint are taken using time-driven approach.

An analysis is also made of the operations involving centralized and distributed checkpoint storage configuration. Fig. 3.12 the checkpoint creation time on the Ephemeral local disks and its replication are shown. The checkpoint creation time on the EBS centralized storage is also presented. It is remark that configuring a centralized approach doubles the time of protection obtaining by configuring FTM to use local storage instead of centralized, even with the distribution operation required by the local storage approach.

The protection configuration is important because it not only directly affects the application throughput, though it also has influence in the MTTR. As the checkpoint time is similar to the restart time, the larger the checkpoint time, the longer time it will be needed to repair (MTTR). In both scenarios, though, the MTTR has a bounded

and limited value, which is dependent on the selected FTM configuration. Figure 3.12, clearly shows that applying the FTM + distribution configuration reduces the checkpoint time, compared to the centralized approach, obtaining higher application throughput and improving the MTTR.

3.3.6 FTM Automatic Restart Evaluation

The automatic restart feature of FTM is evaluated in this section. This evaluation uses the AOCLSB cluster. The applications Matrix Multiplication and NAS-CG benchmark are configured to run 40 and 32 processes respectively distributed in 5 nodes of the cluster.

The restart time is further analyzed for both applications. The Fig. 3.13 shows the time spent to restart the applications from a checkpoint and in two different scenarios: (i) the application restarts from a spare node, plus the time spent to transfer checkpoints to to the spare node; (ii) the application restart using the remaining resources after the failure. As expected, restarting in a reduced resources environment, gives longer restart time and the application runs with less performance. Meanwhile, when the spare node is used, the restart time is shorter, though there is an additional cost to transfer the checkpoint files to the node, which is dependent to factor that has influences on the checkpoint transfer, such as network or storage devices I/O performance, among others.

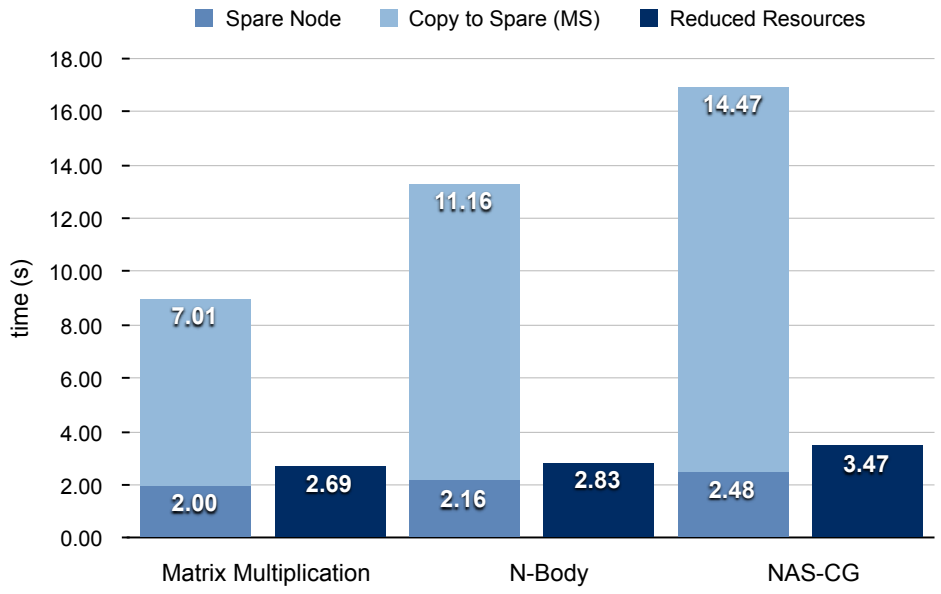


Fig. 3.13 Restart time: Matrix multiplication and N-Body applications executed in AOCLSB cluster.

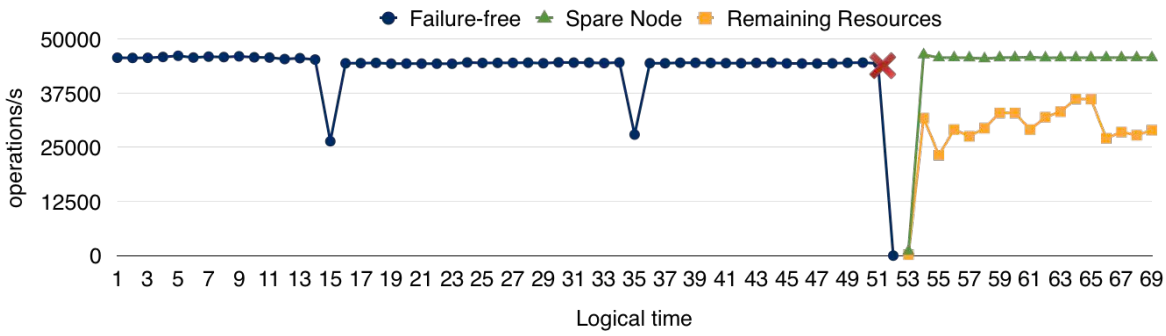


Fig. 3.14 Restart Time: Using AOCLSB cluster.

A full execution scenario including a failure is used to analyze the main factors that influence the MTTR in a parallel execution. The N-body application with the previously seen configuration is used. The Fig. 3.14 depicts an overview of the application execution in terms of throughput (operations/seconds). The throughput is measured in one process of the application execution. The figure shows the fall in the throughput when checkpoints are been taken. When the failure affect the application, it is possible to observe that the throughput is 0. From this point the FTM operations to recover the execution are performed. We plot both recovery alternatives in the same figure: recovery using spare node, and the recovery using the remaining resources. It is noticeable that using the remaining resources overload the execution environment oversubscribing the failed processes. Once again it is expected that final execution time will be degraded by the loss of overall performance.

A detailed calculation can also be done to understand which variables are the most significant for the MTTR. Table 3.4 shows two scenarios for the MTTR calculation, using a spare node to resume the execution and using the remaining resources. As it is possible to observe the major influence in the MTTR calculation is the re-execution time (RX_{time}), which depends on the failure point during the application execution. When configuring large checkpoint intervals, bigger are the probabilities for longer re-executions in case of failures. The worst case scenario, would be the re-executing the whole checkpoint interval. Factors such as checkpoint restart time (CR_{time}), the transfer to spare (MS_{time}) or D_{time} are dependent to the FT implementation. We offer

Table 3.4 MTTR calculation values in the AOCLSB cluster.

Configuration	D_{time}	CF_{time}	MS_{time}	CR_{time}	RX_{time}	$MTTR$
Spare node	3.28	0.02	2.54	1.30	53.81	60.94
Remaining resources	1.36	0.02	0.00	1.54	83.99	86.90

an automatic FT approach, with a controlled detection time, not requiring human intervention.

3.4 Summary

In this chapter, a Fault Tolerance Manager is presented, to provide high availability to users in an automatic and transparent manner. The FTM extends RADIC architecture to support coordinated checkpoints and it adds a facility to replicate and distribute checkpoint files created on node local disks to logical neighbors nodes. The correct functionality of the proposed design is shown. Furthermore, it is tested in a private cluster and in a cloud environment.

We have also seen how different configurations of FTM affects the user's executions. Choices such as setting up a spare node, selecting a centralized storage for checkpoints or a local approach, have impact on the expected execution time and the MTTR value.

The experiments shown in this chapter, allows to verify the functionality of the FTM approach and evidence low impact on the user's execution. In the experiments, it is possible to notice the importance of the FT configuration and how it affects the executions. We can also conclude that the distribution and replication facility built with FTM that allows the automatic recovery has less protection impact comparing it with the traditional centralized approach in both execution environment.

With this work, users can select among the different FT configuration options, knowing in advance how it will impact their expected execution time in failures scenarios, and with this knowledge make the decision that suit best the trade-off among protection costs and high availability.

Chapter 4

Application-Layer FT with Dynamic Resource Controller

"Sometimes life hits you in the head with a brick. Don't lose faith."

Steve Jobs

4.1 Introduction

In the Chapter 3, the Fault Tolerance Manager (FTM) was presented for coordinated protocols, although the architecture design is extensible, and is possible to support uncoordinated and semi-coordinated rollback-recovery protocols.

In this chapter, we leverage the FTM to the application-level, offering automatic and transparent mechanisms to recover applications in case of failures, meaning users are not required to perform any action when failures appear. The solution uses semi-coordinated and uncoordinated rollback-recovery protocols following RADIC architecture. FTM combines application-level checkpoints with a sender-based message logger using the concepts of ULFM for detection and recovery purposes. Furthermore, a dynamic resource controller is added, which performs the monitoring of main memory usage for the logger facility, allowing to detect when its usage is reaching a limit. With this information, it invokes automatic checkpoints, in an optimistic manner, which allows freeing memory buffers used for the message logger avoiding the slowdown or stall of the application execution.

The content of this chapter is organized as follows: The design of FTM in the application-level is presented in 4.2. The dynamic resource controller functionality is

described in 4.2.2. The experimental evaluation, which contains the FTM functionality validation and the dynamic resource controller verification with a well-known NAS benchmark is shown in 4.3. Final remarks are stated in 4.4.

4.2 FTM in the Application-Level with Dynamic Resources Controller

This section describes the design of the Fault Tolerance Manager (FTM) in the application-level to provide high availability to the user's applications in an automatic and transparent manner. The traditional stack for HPC execution environment is composed of several failure prompt layers. FTM global solution isolates the user's application layer from failures. It suits the execution environment with a handler controller, which deals with failures recovering the user's application execution when failures appear. The solution components are depicted in Fig. 4.1.

The Fault Tolerance Manager is composed of a sender-based message logger, which combined with the application-level uncoordinated and semi-coordinated checkpoints, protects the application during failure-free executions. A dynamic resource controller is attached, which monitors and manages FTM resource usage for FT protection.

4.2.1 FTM in the Application-Level

FTM follows both the uncoordinated and semi-coordinated rollback-recovery protocols. The application state is saved using application-level checkpoints and the exchanged messages are stored in a sender-based message logger. Following this approach, only the information regarding the application is persisted. In the following subsections the design of the FT tasks for FTM are explained:

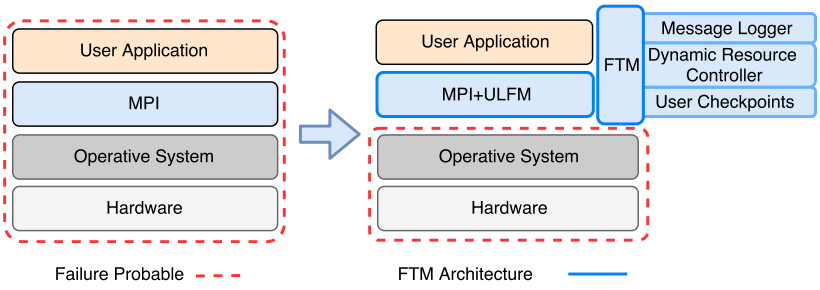


Fig. 4.1 FTM in the Application-Level.

Protection

The fault tolerance protection is done by taking checkpoints of the application processes and storing exchanged messages between processes in a sender-based message logger facility, during the application execution. The components are described as follows:

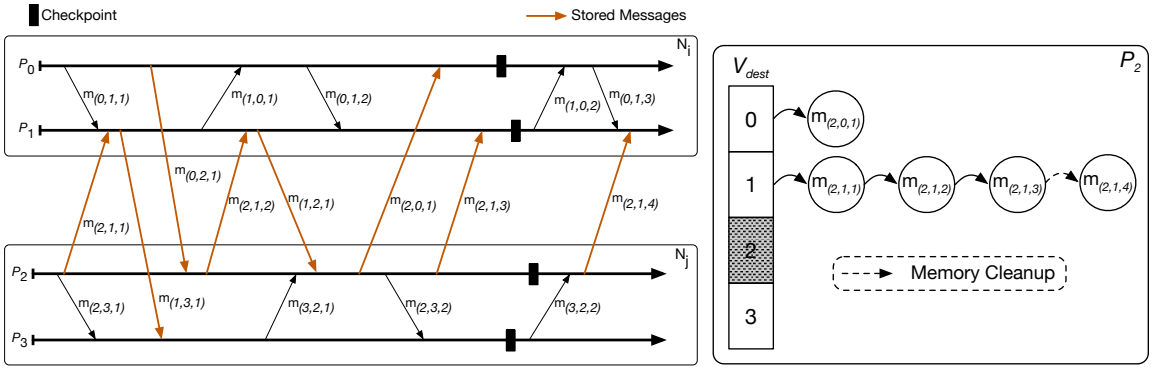
Application Checkpoints The checkpoints are taken from the application-level, the checkpointing operation is initiated by the application, hence modifications in the application's source are necessary for the checkpoint invocation, although the application algorithm remains intact. The checkpoints invocation are inserted during natural synchronization of the application processes. The checkpoints store structures containing only necessary information to restore execution in case of failures, avoiding the need to store SO particular information.

Each process creates checkpoints that are flushed into files to the local storage (disk or main memory) for performance reasons. According to the base property of distribution of RADIC and FTM architectures, the checkpoint files are transferred to neighbor nodes, after their creation, to maintain reachability in case of hard node failures. This process is performed on background, minimizing the interference with the application execution.

Message Logging During the application execution, messages are stored into a message logger facility, in order to replay them to the processes that are affected by failures. The uncoordinated and semi-coordinated protocols, have the advantage that only failed processes restart execution using the last healthy checkpoint available, allowing other processes to continue their execution, minimizing the computation waist. After the processes are restarted, during the re-execution, they directly consume messages from the logger facility. FTM in the application-level, uses a pessimistic sender-based message logger. As it works at the application-layer, it is possible to avoid dependencies to a specific MPI library implementation.

For the uncoordinated approach, all exchanged messages between the application processes are stored. The semi-coordinated approach store only exchanged messages between the application processes that are in distinct nodes, as shown in Fig. 4.2a.

To describe the functionality of the Sender-Based Message Logger, the semi-coordinated protocol implementation is used. The logging procedure is done as follows: Each process ($P_1..P_N$) of the parallel application generates m messages, described as: $m_{(i,j,k)}$, where i = source process, j = destination process and k = is the send sequence number. This representation is shown in Fig. 4.2b. The logger instantiate an array of



(a) Semi-coordinated stored messages. (b) Structure Representation.

Fig. 4.2 Sender-Based Message Logger.

linked lists in the node volatile memory, as shown in Fig. 4.2b. The structure allows to store each outgoing message from the processes. For example, P_2 first send $m_{(2,1,1)}$ to P_1, and the logger stores it in the linked list of the array V_1. When P_2 sends $m_{(2,1,2)}$, it is also added in the linked list of the array V_1, putting the message in the tail of the linked list and so on for every outgoing message. It is noticeable that message $m_{(2,3,1)}$ and $m_{(2,3,2)}$ are not stored in the logger because they are inter-node communications of node N_i.

Every time a recovery line is obtained, a memory cleanup is performed. Although, a controller is required to constantly monitor FT resource usage. For e.g.: application processes may be affected due to high memory usage during FT protection, causing the processes stall.

Fault Detection, Reconfiguration and Recovery

When failures appear, a mechanism of detection is needed to start the recovery procedure. For the FTM in the application-level, ULFM is used to detect failures. FTM implements an error handler, which is invoked by the ULFM detection mechanism to recovery the application execution.

In Fig. 4.3, the FTM handler for failure detection, reconfiguration and recovery is depicted. For simplicity, one process per node is assumed, using the uncoordinated protocol with a pessimistic sender-based message logger. The Fig. 4.3, shows that the node running P_3 fails, and P_2 is the first process which acknowledge the failure, which is detected by ULFM, causing the revocation of the global communicator using MPI_Comm_revoke. After the revocation, all remaining processes are notified and they shrink the global communicator, taking out the failed processes using the

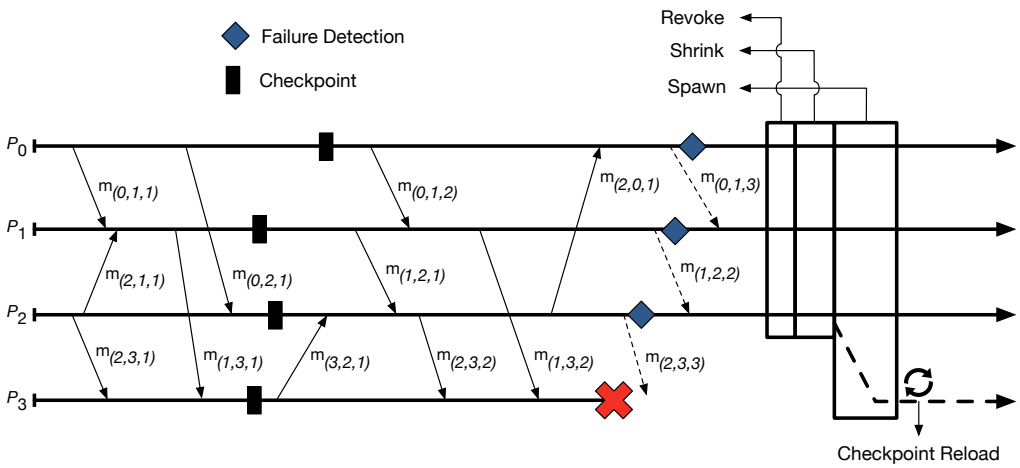


Fig. 4.3 FTM for uncoordinated and semi-coordinated protocols: Example of failure detection, reconfiguration and recovery procedures for one failure. Each message is denoted as: $m_{(i,j,k)}$ where i = source process, j = destination process, k = send sequence.

MPI_Comm_shrink call. Finally, the remaining processes spawn the communicator using a dynamically launched process of the application. The process is re-launched using MPI_Comm_spawn primitive. These operations are collectives, hence depends on the MPI library implementation and its performance are related on the size of the applications processes. Nevertheless, the actions are performed during the fault detection, reconfiguration and recovery, hence not affecting the application scalability.

The spawned process, acknowledges that it has been re-launched and load the checkpoint file. After the checkpoint file is loaded, the variables in the process are set as they were in the last checkpoint prior failure. The process jump to the correct execution line in order to continue the application execution. The messages are consumed from the message logger to finish the re-execution. Meanwhile, non-failed processes can continue their execution.

4.2.2 Dynamic Resource Controller

As previously seen, the FT protection requires resources and it comes with overhead for the user's applications. The uncoordinated and semi-coordinated protocols avoid the restart of all the application's processes when a failure occurs. Although, they require a logger facility to replay messages to restored processes. To reduce failure-free overhead, the logger uses main memory to store processes messages, when enough memory is available, as it often provides higher speed access compared to local hard

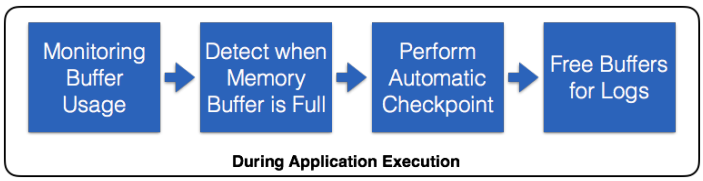


Fig. 4.4 Dynamic resource controller process schema.

disk or a centralized storage. However, the main memory is a limited resource, which is shared between the application processes and the FT components. The usage of FT resources is application dependent, meaning different communication pattern, size and quantity of messages, directly impacts on FT resource usage. The available memory can rapidly ran out due to FT protection tasks. The impact of running out of main memory can result in the application execution become stalled.

FTM performs the protection of the application execution, storing both: checkpoints, and messages of the application processes. Although, no control is performed regarding the the resource usage during FT protection. In order to avoid free memory ran out due to FTM protection task, a dynamic resource controller is introduced with FTM. It works on each node of the application execution.

The dynamic resource controller constantly checks the currently allocated memory of the array of lists structure used for message logging purposes (described in the previous section). When it detects that usage is reaching the available limit, it triggers an automatic checkpoint invocation, which consequently stores the state of the application and frees the used memory buffers for logging purposes, providing the application FT protection and at the same time avoid to interfere the application's memory usage (Fig. 4.4). Interval based checkpoints are not affected, and when they are processed, the memory buffers are also released.

As the monitoring is performed at the same time as the message storage into the message logger, no additional overhead is expected. Although, when automatic checkpoint are invoked to free-up memory usage, the cost of each snapshot is introduced.

During the application execution the memory may ran out meaning the lost in terms of performance of the application execution. The controller detects it and automatically invokes a checkpoint creation of each application process, when they reach the next natural synchronization point, allowing to free memory usage by the logging facility, therefore avoiding the application's execution stall due to the lack of main memory.

To illustrate the dynamic controller functionality, an example is shown in Fig. 4.5. There is one process running per node $P_{(0...3)}$, predefined checkpoint invocation are configured as well, called default checkpoints. The controller constantly monitors

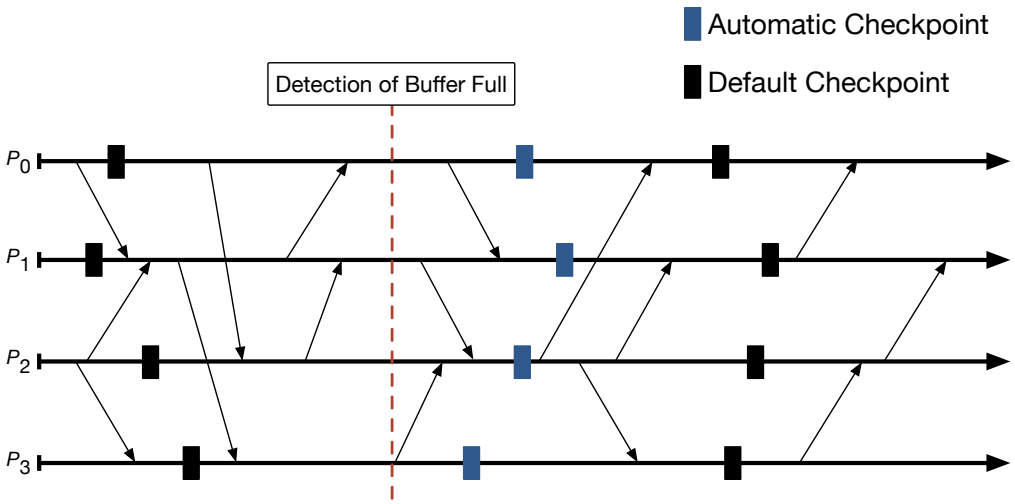


Fig. 4.5 Functionality example.

the message logger usage. During the application execution, P_3 detects that the buffer reaches the usage limit, hence schedule an automatic checkpoint invocation. The checkpoint is done in the next natural synchronization point of the application execution.

4.3 Experimental Results

This section presents experimental results obtained applying FTM to provide Fault Tolerance in the application-level. The results show its automatic functionality and verifies the functionality of the dynamic resources controller in real execution environments. The experiments were developed in controlled environments using AOCLSB-FT and AOCLSB-L clusters and injecting failures. Executions are performed at least 3 to 5 times, unless it is explicitly specified differently, and measurements are taken using the ***time***, system tool.

The clusters basic configuration are shown in Table 4.1, and further details are described in the Appendix A.0.1. For the experiments performed in this section, the clusters are setup with Open MPI implementation with the ULFM 1.1 extension, which is configured with **–with-platform=optimized** option, to avoid interference of debugging inside the MPI library in the experimental results.

The application used in the experiments is the NAS CG Benchmark. The experiments are performed using Class C and D which have 150000 and 1500000 rows respectively.

Table 4.1 Experimental environments.

COMP.	AOCLSB-FT	AOCLSB-L
CPU	2 quad-core Intel(R) Xeon(R) E5430 @ 2.66 GHz	8 AMD Opteron (x8) Processor 6262 HE @ 1.60 GHz
RAM Memory	16 GB	252 GB
Local Storage	30 GB HDD	
Operative System	CentOS release 6.4	CentOS release 6.2

To implement the FTM protection, the application's source code has been modified to add the checkpoint invocation in a natural synchronization point. The checkpoints are taken at 50% of the application execution.

4.3.1 FTM Performance and Cost Evaluation

In this section the validations of FTM functionality is done using the uncoordinated and semi-coordinated rollback-recovery protocols along with a pessimistic sender-based message logger. The application is the NAS benchmark CG, and to show scalability behavior, 3 different processes quantity are used: 16, 32 and 64.

The performance of FTM is tested applying it to the CG application. During the experiments a failure is injected to one node of the cluster at 75% of the application execution. In order to analyze the benefits of applying the solution, a *Reference Time* is calculated. It represents the scenario in which a failure is injected at 75% of the application execution, as checkpoint is made at 50% of the execution, we assume that 25% of the execution is lost due to the failure. To finish the application execution, is equivalent to a total execution time of around 125%, due to the lack of FT protection nor FTM. This time compared to the measured time of executing the application with FTM protection experimenting a failure (FTM w/Failure).

Figure 4.6, shows the results of the execution time normalized to the execution without Fault Tolerance (No-FT). The experiments were done using 16, 32 and 64 processes with the CG Class C using 9 nodes of the AOCLSB-FT cluster, 8 for the application execution and 1 as spare node. It is possible to observe that having FTM protection (FTM w/Failure) saves user time approximately 13% compared to the *Reference Time* when a failure appears, for the scenario where 64 processes of the CG Class C application are used.

The cost of applying FTM is composed of two main elements, the checkpointing operation and the message logging. The Fig. 4.7a shows that FTM protection

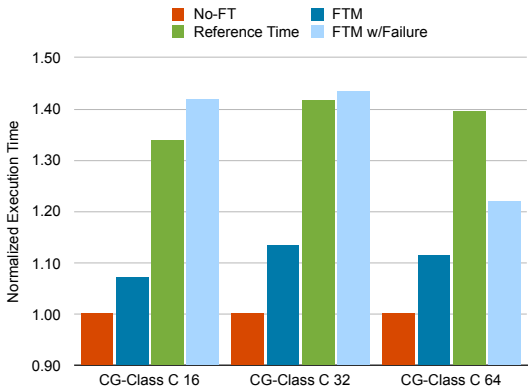
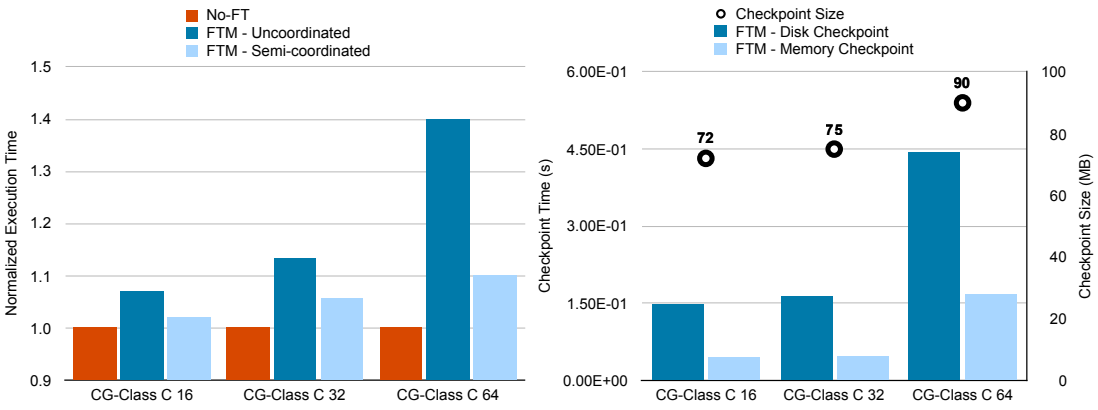


Fig. 4.6 CG-Class C with FTM with the uncoordinated checkpoint protocol, in a failure injection scenario using the AOCLSB-FT cluster.

costs are between 5% and 40%, depending on the amount of processes, quantity and size of messages and communication patterns. Although, the semi-coordinated FTM implementation drastically reduces this cost, avoiding to store internal messages between processes on running on the same node. The benefits for the 64 process application is the protection costing 10% overhead.

Another key factor for the protection cost is the storage. To illustrate this factor, FTM is setup to store checkpoints and message logs into main memory, though we measure the checkpoints usage and compare against saving them to local disk. The experiment results are depicted in Fig. 4.7b. It is noticeable that, using main memory to store the checkpoints files, is at least 2.5x times faster than using the disk when available, and the FTM can benefit from its performance.



(a) Execution overhead. (b) Checkpoint times and sizes.

Fig. 4.7 FTM in the application-level costs.

4.3.2 Dynamic Resource Controller Evaluation

In this section the evaluation of the dynamic resource controller is performed. For the experiments, two scenarios are illustrated, one where there is enough memory resource on the execution environment for both, the application and the FT protection. The second scenario, where the execution environment does not have enough resources for the FT protection, and an action must be taken in order to keep the application running.

An experiment is performed running CG with a larger workload (Class D), on 4 nodes of the AOCLSB-L, which has significantly more resources compared to the AOCLSB-FT. In Fig. 4.8, the throughput of the application is shown. It is noticeable that the application throughput is almost constant, we can state the FTM with its logger does not significantly affect the application throughput. Although, at the 50 iteration (50% of the application execution), there is a noticeable fall-down in the application throughput, due to checkpoint creation during the execution.

The Fig. 4.9 illustrates the memory usage for the sender-based message logger of one process during the application execution. It is possible to notice, that the memory consumption grows due to the availability of the resource, and when a checkpoint is taken, the memory buffers are released. The dynamic controller constantly monitors the memory availability, and in this particular case, perform no actions, because it detects that there is enough memory for the logger, prioritizing storage performance for the FT protection, as the application will not be affected.

Meanwhile, running the CG Class D application in the AOCLSB-FT cluster, which has less resources compared to the AOCLSB-L, and configured to take one checkpoint at 50%, as in the previous experiment, shows the dynamic controller performing actions. Two scenarios were evaluated, with and without the dynamic controller.

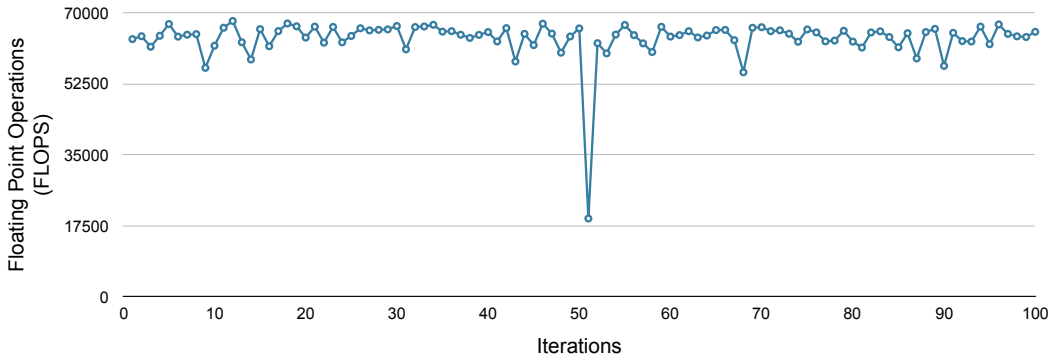


Fig. 4.8 Application throughput.

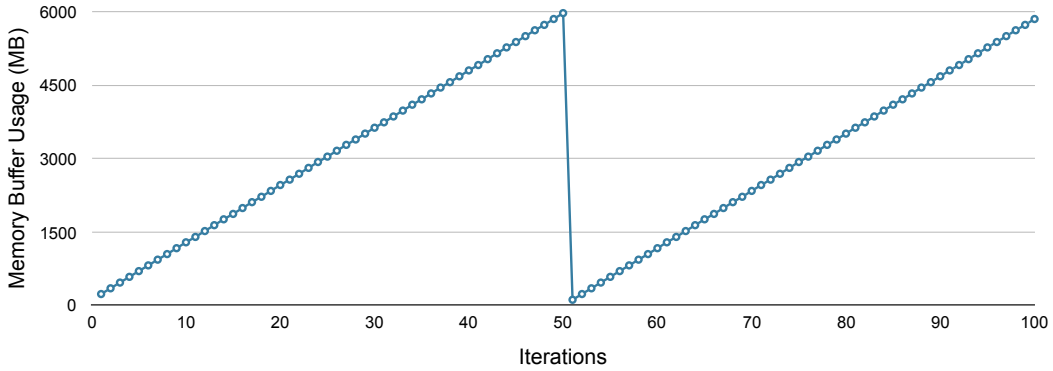


Fig. 4.9 Memory usage monitoring.

Figure 4.10, shows both executions starting with a similar application throughput, though when the execution without the dynamic controller starts using SWAP memory zone of the system, the throughput drastically drops, making the whole application crash. Meanwhile, the execution with the dynamic controller, optimistically invoke the checkpoints, and after their completion, the memory buffers used for the logger facility are released, allowing the continuous execution of the application by not affecting the main memory available for the application processes. This scenario was designed to use the low resources cluster and an application with high FT resources demand, to visualize the behavior of the application with and without the dynamic resource controller, and that is one of the reasons for several checkpoints creation by the dynamic controller, in order to keep execution going, and deliver the application results.

It is important to remark that the dynamic controller does not interfere in the user-defined checkpoint events, letting users the control of the checkpoint moments,

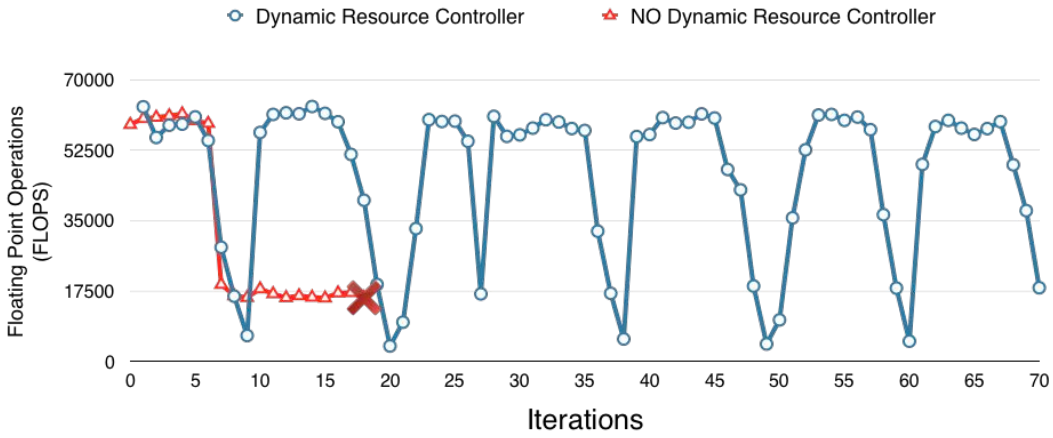


Fig. 4.10 Application process throughput.

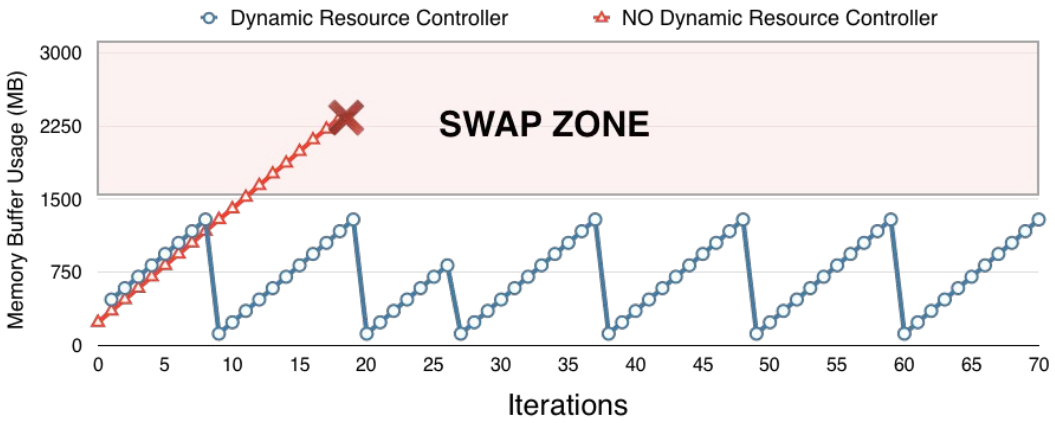


Fig. 4.11 Memory usage monitoring.

though it may perform optimistic checkpoints, to free resources for the application. The solution allows the application to continue the execution, though it may come with larger overhead, due to the automatic checkpoints invocation. Figure 4.11 shows how the memory is managed during the application execution in contrast to the execution without the management.

Finally, the FTM plus the dynamic controller allows the application to finish, providing high availability and offering multiple configuration possibilities.

4.4 Summary

This chapter presents FTM in the application-level. It describes how FTM architecture shown in (Chapter 3), can be easily extended to support uncoordinated and semi-coordinated rollback-recovery protocols. The uncoordinated and semi-coordinated protocols implementations offers the capability of restoring only affected processes in failure situations, optimizing the user's application execution time in failure scenarios.

This proposal contributes providing a novel Fault Tolerance Manager in the application-layer, allowing users to define only the necessary protection information for their applications. Furthermore, a dynamic resource controller is suited to FTM, which monitors FT resource usage and perform actions when the usage reach boundaries where it may affect the application execution.

Chapter 5

Configuring Fault Tolerance Protection and Recovery

"There's a way to do it better - find it."

Thomas A. Edison

5.1 Introduction

Fault Tolerance (FT) techniques must be applied to HPC systems in order to ensure high availability to parallel applications execution with a minimal cost in terms of resources and overhead. As seen in previous chapters, rollback-recovery protocols are widely used within FT to protect application executions [30].

Actually, the checkpoint interval is one of the main factors that determines overhead added to the application failure-free execution. Models to define an optimal interval have been created for at least 35 years [34], mainly to reduce the overhead added to the applications. The goal with the models is to find the optimal trade-off where the protection does not interfere with the progress of the execution, and at the same time, protect the application enough, hence in case of failure the degradation would not be so high. Figure 5.1, shows that when the checkpoint interval is large, more work can be lost in case of failures, meanwhile, smaller intervals avoid losing work, but at the same time comes with higher overhead.

When a hard failure appears on a computational node, FT mechanisms, usually recover the execution by re-configuring the execution environment using a spare node, and with the new environment, resume the application execution. If a spare node is

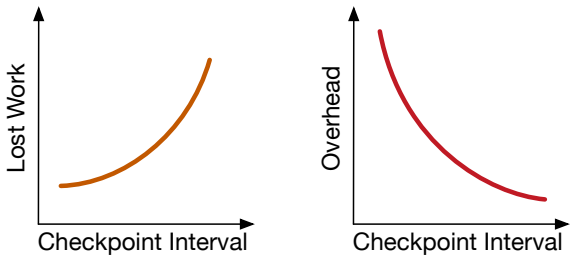


Fig. 5.1 Checkpoint interval trade-off.

not available, and a failure occurs, the FT mechanisms try to resume the execution using only the remaining resources. A known problem in the fault tolerance topic is to be able to determine if an application can continue its execution when no spare resource is available [61].

In this chapter, two models are presented for FT protection and recovery of parallel applications: a First Protection Point (FPP) model is introduced, which allows determining initials not necessary checkpoints given an optimal checkpoint interval calculation, and; a Spare Node model that evaluates; the possibility to recover the application using remaining resources, and the performance loss factor of a parallel application when it lost computational nodes, and continues its execution without using spare node resources. It also allows to detect when there are not enough resources to continue the application execution forcing a safe-stop action.

The FPP model studies the effects of FT tasks such as protection and recovery to elaborate the model, and basically determines that depending on the moment in which a failure is experienced in parallel applications, it is more convenient to completely restart the application, than restarting it from a checkpoint, in terms of total execution time. As a checkpoint interval model determines points in the application where a checkpoint should be made, this FPP model focuses on avoiding initial checkpoints, that leads to an overhead reduction. Meanwhile, the Spare Model, helps to identify the effects of continuing an execution using remaining resources when recovering from a failure in a particular point during the application execution. This allows to decide weather to acquire or not the spare node when a failure occurs, improving resource usage.

In the following sections, the FPP and the Spare Node models are described for uncoordinated and coordinated rollback-recovery protocols. An analytic evaluation of both models is done. Also, experimental evaluation are made, to show use-cases of the proposed models.

5.2 First Protection Point Model

Provide high availability to users applications comes with a considerable overhead. The work introduced in this section, tries to minimize this factor designing a FPP model for FT protection based on checkpoints.

The main goal of this model is to help users determining the point in their application execution to start taking checkpoints, based on the assumption that before that point, re-executing the whole application after a failure is less expensive than it would be by protecting with checkpoints and then perform a restart. Furthermore, with FPP it is possible to explicitly avoid checkpoint invocations that are defined by a checkpoint interval model.

It is important to remark that the FPP model is designed taking into account the first possible failure that may occur during the application execution. The model also assumes that probability of failures is uniformly distributed over the whole application execution.

5.2.1 Designing the First Protection Point Model

Characterizing total application execution time, using main FT protection factors that have significant effects on application execution allows to obtain a k point during application execution, which determines the starting point where FT protection is effective. The values that k can take are between $0 \leq k < 1$, and it represents a percentage of the application execution time.

Supposing an application, which runs without FT protection, the total time to finish its execution is ET_{nft}. If the application experiences a failure, management tasks are executed in order to start the application from the beginning. t_{mgmt} is the time to perform management tasks. RET_{nft} (Eq. 5.1) is the total time to finish remaining execution after failure and ET_{nft-f} represents the total execution time of the application without FT, experiencing a failure. The scenarios are depicted in Fig. 5.2.

$$RET_{nft} = t_{mgmt} + ET_{nft} \tag{5.1}$$

To enable FT protection, checkpoints that can be: coordinated, uncoordinated or semi-coordinated are stored into a stable storage. Depending on the protocol used for checkpointing, message logs are also stored. For uncoordinated protocols, the FT

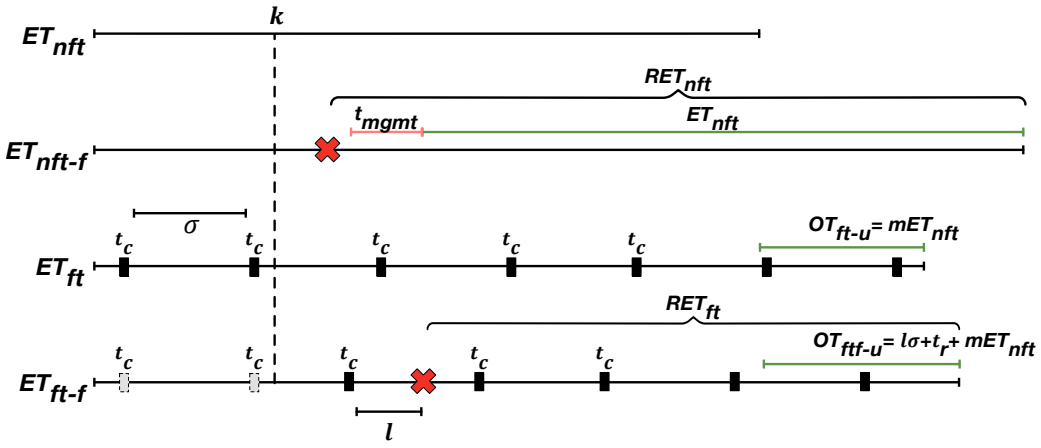


Fig. 5.2 Modeling scenarios for FPP model.

overhead can be defined as the total amount of time spent taking checkpoints, plus the consumed time for message logging. Storing the state of the application processes comes with an overhead m, which is added over the application execution time without FT (ET_{nft}). The FT overhead is denoted using the execution time with FT (ET_{ft}) and without FT (ET_{nft}) in Eq. 5.2.

$$m = \frac{ET_{ft} - ET_{nft}}{ET_{nft}} \tag{5.2}$$

Having the application protected with a checkpoint interval σ, a checkpoint is taken periodically and the time consumed on each checkpoint is t_c. When a failure appears during application execution, the point of failure can be set in reference to the checkpoint interval. The executed portion of checkpoint interval till a failure is defined as l (Fig. 5.2).

$$RET_{ft} = l\sigma + t_r + [(1-k)mET_{nft}] + (1-k)ET_{nft} \tag{5.3}$$

Remaining execution time after a failure RET_{ft} is composed of the portion of execution lost due to failure ($l\sigma$), plus the checkpoint restart time (t_r), the remaining execution time $(1-k)ET_{nft}$, and the time added due to overhead caused by FT protection after point k: $[(1-k)mET_{nft}]$. Total application execution time with FT protection and including failure is ET_{ft-f}. This scenario is illustrated in Fig. 5.2.

Point k definition

The k value represents the starting point in parallel application executions where FT protection (checkpoints + logs) becomes effective. The condition $RET_{nft} > RET_{ft}$ has to be verified in order to determine the point k. Using this condition, the k value can be obtained:

$$ET_{nft} + t_{mgmt} > l\sigma + t_r + [(1-k)mET_{nft}] + (1-k)ET_{nft} \tag{5.4}$$

$$k > \frac{l\sigma + t_r + mET_{nft} - t_{mgmt}}{mET_{nft} + ET_{nft}} \tag{5.5}$$

Coordinated protocols

We can elaborate a more accurate approximation regarding the FT protection overhead for k point value calculation, when using the coordinated checkpoints. The total amount of checkpoints for an application execution can be defined as n (Eq. 5.6).

$$n = \frac{ET_{nft}}{\sigma} - 1 \tag{5.6}$$

In order to re-write Eq. 5.3, which determines the remaining execution time after a failure appears, to make overhead more accurate for coordinated protocols, it is possible to calculate the amount of checkpoints that are going to be made, after a failure occurs. A number of checkpoints after failure is calculated n_{af} using the Eq.5.7.

$$n_{af} = \frac{(1-k)ET_{nft}}{\sigma} - 1 \tag{5.7}$$

Using the new overhead calculation the Eq. 5.3 is updated to Eq. 5.8, which is used to obtain the k value.

$$RET_{ft} = l\sigma + t_r + [\frac{(1-k)ET_{nft}}{\sigma} - 1]t_c + (1-k)ET_{nft} \tag{5.8}$$

Upgrading the Eq. 5.5, using the Eq. 5.8, to make use of the accurate overhead value for coordinated protocols gives the following Eq. 5.9 to determine k.

$$k > \frac{l\sigma^2 + t_r\sigma + t_c(ET_{nft} - \sigma) - t_{mgmt}\sigma}{(\sigma + 1)ET_{nft}} \tag{5.9}$$

5.2.2 First Protection Point Analytical Evaluation

To evaluate FPP model using an analytic approach, a series of example values are developed for an uncoordinated protection shown in Table 5.1. First, k value is calculated to define the starting point to insert FT protection. Then, the execution time after failure (RET_{ft}) is obtained, simulating failures in different points of the execution. Failure locations vary in values that are after, before and on the calculated k, in order to verify if FT protection is effective considering execution time after failure.

We consider that no time is expended regarding the management time after a failure occurs ($t_{mgmt} = 0$). This consideration if far from reality when human intervention is needed, as the failure detection, system re-configuration, and application re-launch certainly takes more than 0 seconds. Although, this evaluation uses the worst case scenario, showing where to start FT protection in an effective way. Furthermore, assuming that failures are in the middle of checkpoint intervals (σ), $l = 0.5$ is used.

Using parameters from Table 5.1 the value of k is 0.32 (32% of the application execution), which was calculated with the FPP model for uncoordinated protection (Eq. 5.5). This value represents the inflexion point where the FT protection becomes effective (Fig. 5.3) . For this example, the k value point to 3200 seconds after the

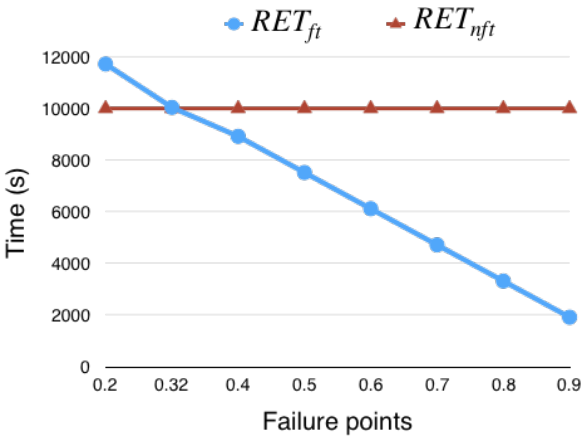


Fig. 5.3 Execution time after failure.

Table 5.1 Parameters for k value calculation.

Variable	Values
m	0.4
l	0.5
ET_{nft}	10000 seconds.
t_r	20 seconds.
σ	1000 seconds.

execution starts ($k * ET_{nft} = 0.32 * 10000 = 3200$). A failure that occurs in a point before k will make a FT protected application ends its execution in more time compared to re-executing the whole application without FT. It is important to recall that, after the first failure no other failure is contemplated for this analytic scenario.

5.3 Spare Node Configuration Model

Spare node configuration represents an important issue of the FT configuration for a parallel application execution. The main goal of the Spare Node Configuration Model (SNCM) is to help configure spare resources for a FT protected parallel execution.

Using the FTM protection, an evaluation of memory and storage usage for FT proposal is done, which helps to define if the applications can run without using spare resources, and still warranting the application completion despite failures.

The SNCM is used to evaluate if an application can run without a spare node and still be protected against failures, offering a limited MTTR. SNCM characterizes the application execution with FT protocols and available resources to obtain the main factors that are applied in the presented model. Furthermore, SNCM is designed to determine what is the overhead and time cost value of an user's application execution experiencing a failure in a moment during its execution. Moreover, the proposal determines that when a failure occurs, resuming the execution can be done using the remaining resources without losing performance.

The SNCM model uses a performance-loss factor, which represents the impact of losing computational resources when a failure appears. The model allows to design a procedure to configure spare node for parallel executions. The procedure is illustrated in Fig. 5.4.

In the following subsections, the memory and storage evaluation design are described, which are used for the spare node configuration. Then, the SNCM is defined and an analytic evaluation of the model is done.

5.3.1 Fault Tolerance Storage Evaluation for Recovery

The memory and storage evaluation of the user's applications is performed to determine if the application can run without a spare node and, in case of failures, remain running as a result of FT protection. Then, if a failure appears the application can be recovered using only the remaining resources.

As main memory is often the fastest storage resource available in a computer node, trying to take advantage of the main memory often represents less protection time, reducing the overhead of the FT. A favorable situation would be to run the application in an environment that has enough memory resources to run the application processes and also has enough memory for (local and neighbor) process checkpoints files. The checkpoint files may also be compressed prior storage.

The evaluation is done by characterizing the user's application on the system environment, applying FT. The tools used for the characterization are linux performance observability tools, such as: *time* for time statistic of the processes, *free* for memory availability in the nodes, *ps* for process status, Virtual Set Size (VSZ) and Reserved Set Size (RSS) memory usage. The tools are defined in the Single Unix specification [70], and implemented in the OS distribution. Scripts are also designed and used for monitoring purposes. After characterizing the application execution with FT protection on the system environment, it is possible to calculate the amount of memory needed on the nodes to protect the application with FT, so in cases of failures, recovery the lost processes on the remaining resources.

The Fig. 5.5, illustrates a diagram, which is used to evaluate the FT storage configuration. It is used to determine if a spare node is not mandatory for restoring the execution, the local nodes need to have enough free memory to host processes from its neighbor node, hence in case of failures the execution could continue after recovery. The analysis continues by determining what is the best resource option to store the checkpoint files from local and neighbor processes. The result of the

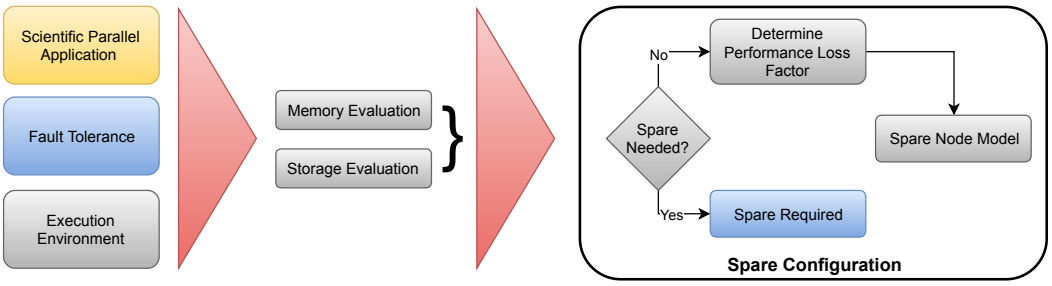


Fig. 5.4 Spare Node configuration procedure.

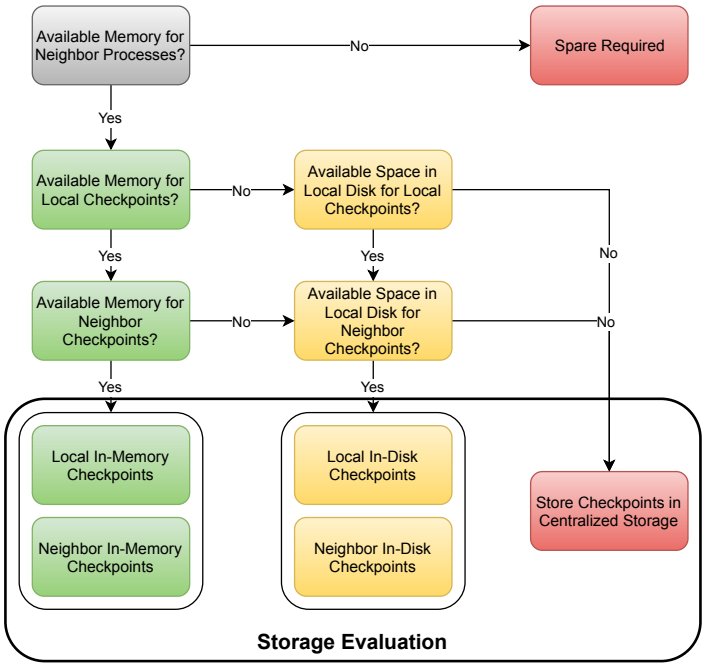


Fig. 5.5 FT Storage evaluation for recovery.

evaluation process determines the configuration of an application execution protection in a specific execution environment.

The FT storage evaluation aims to keep protection information in the fastest available device. It also helps determining if the FT protection for the application must be configured with spare node. The recovery approach follows the FTM recovery model, which distribute protection information among the neighbors. When no spare is available, FTM recovers the application using remaining resources. When the recovery is not possible, a safe stop of the user's application is done.

5.3.2 Designing the Spare Node Configuration Model

The Spare Node Configuration Model explores the effects of FT protection and recovery to the execution time of the parallel applications. Particularly, the model aims to determine the effects of the loss of computational resources in the recovery process of the FT. The main objective is to obtain the moment during the application execution where recovering the application using only the remaining resources is better than acquiring a spare node to continue the application execution.

In order to develop the model a characterization of total application execution time using FT protection and of the recovery factors is made. The characterization allows modeling the application behavior during failure and failure-free executions. Figure

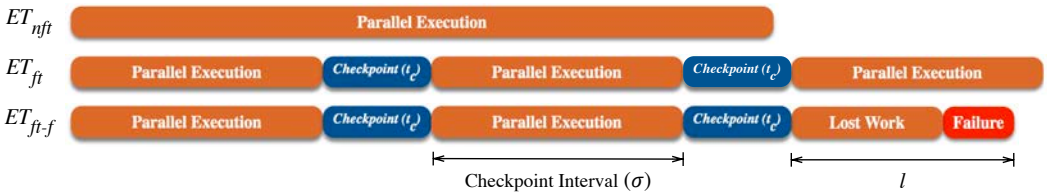


Fig. 5.6 Application execution scenarios.

5.6 depicts an application execution without FT and the same application with FT. It also shows a failure scenario in which a portion of the work is lost due to a failure.

As seen in section 5.2.1, the overhead depends on the time expended on taking checkpoints and performing message logging, and is defined in Eq. 5.2. Having the application protected with FT, hence taking checkpoints in intervals (σ), and the checkpoint time of each process is t_c. Similarly to section 5.2.1, when a failure occurs, the point of failure can be set in reference to the checkpoint interval σ (Fig. 5.6).

After a hard node failure occurs, there are two options to resume the execution, both are shown in Fig. 5.7. If a spare node was reserved, processes checkpoint files of the lost node have to be transferred to the spare node, in order to perform a restart. The time to copy the checkpoint files is considered as t_{cs} and the time to perform restart using a spare node is t_{rs}. The total time to finish remaining execution after failure using a spare node is RET_{ft-s} (Fig. 5.7).

Restarting the application using the remaining resources, can avoid the checkpoints transfer operation, depending on the FT storage configuration according to the model described in 5.5. Although, using remaining resources, overloads one node with the processes of the failed neighbor node. The time to perform restart from checkpoints in this scenario is t_{rr} and RET_{ft-r} is the total time to finish remaining execution after failure. When remaining resources are used to resume execution, it is expected to have a performance loss, due to the missing processors of the failed node. The performance loss factor is defined as γ (Eq. 5.10), and it is illustrated in Fig. 5.7.

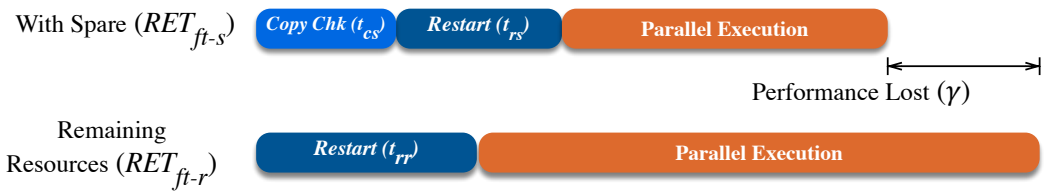


Fig. 5.7 Spare Node model: recovery alternatives.

$$\gamma = \frac{RET_{ft-s}}{RET_{ft-r}} \tag{5.10}$$

Point s definition

For the s point definition we suppose an FT protection using FTM. As defined in Chapter 3, it distributes checkpoint files along the compute nodes, hence in case of failures, it is possible to automatically restart the application. This FT storage configuration is shown in Fig. 5.5.

The s value represents the starting point in the parallel application execution where it is convenient to resume execution using the remaining resources in failure scenarios. The values that s can take are between $0 < s < 1$, and it is a percentage of the application execution time. Figure 5.8 depicts the point s where the not using spare nodes is more effective than using them. The overhead is the key factor analyzed to define the s point.

The s point is obtained to determine that after a failure occurs, resuming the execution using remaining resources is equal or smaller in terms of execution time, compared to setup a spare node and resume execution using it. This scenario may happen if the transfer or the setup of the node is a costly operation. To obtain the s value, the following condition (Eq. 5.11) has to be verified:

$$RET_{ft-s} > RET_{ft-r} \tag{5.11}$$

The condition 5.11, requires the definition of the remaining execution time after failure is made for both cases: with a spare node (RET_{ft-s}) and using the remaining resources (RET_{ft-r}).

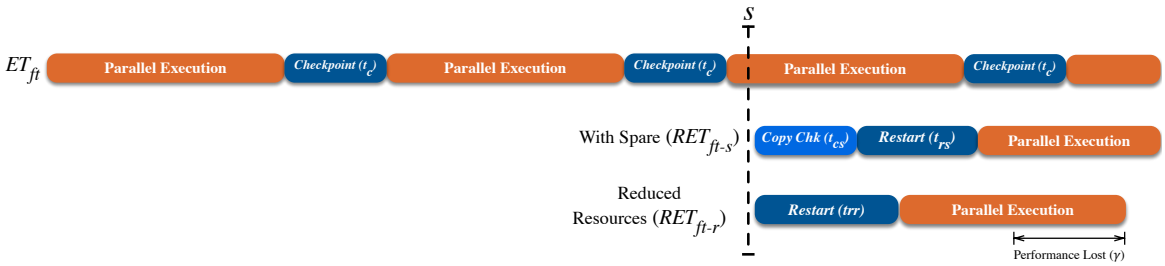


Fig. 5.8 Point s Schema.

Remaining execution time after a failure using a spare node RET_{ft-s} (Eq. 5.12) is composed by a portion of the lost execution due to failure ($l\sigma$), plus a time to copy checkpoints to the spare node (t_{cs}), a time to perform restart from checkpoints (t_{rs}), the remaining execution time $(1-s)ET_{nft}$, and the time added due to the overhead caused by FT protection after s point: $[(1-s)mET_{nft}]$.

$$RET_{ft-s} = l\sigma + t_{cs} + t_{rs} + [(1-s)mET_{nft}] + (1-s)ET_{nft} \tag{5.12}$$

Recovering execution after a failure using only the remaining resources, can avoid the checkpoint copy operation, although the performance is affected due to the loss of computing resources. The Eq. 5.13 defines the remaining execution time after a failure appears using remaining resources.

$$RET_{ft-r} = \gamma l\sigma + t_{rr} + \gamma(1-s)mET_{nft} + \gamma(1-s)ET_{nft} \tag{5.13}$$

Finally, the s value model (Eq. 5.14) is determined using the condition Eq. 5.11, with the values defined in the Eq. 5.12 and 5.13, the s point value is cleared as:

$$\begin{aligned} & s > 1 + \beta \\ & \text{where:} \\ & \beta = \frac{l\sigma(\gamma-1) + t_{rr} - t_{cs} - t_{rs}}{ET_{nft}(a\gamma - a)} \\ & a = 1 + m \end{aligned} \tag{5.14}$$

Table 5.2, shows the variables used to develop the model and its corresponding descriptions.

Table 5.2 Variables description.

Variable	Description
σ	Checkpoint interval
m	Overhead percentage
l	Executed portion of σ in which a failure is detected
ET_{nft}	Execution time without fault tolerance
t_{rr}	Time to restart application using remaining resources
t_{rs}	Time to restart application using a spare node
t_{cs}	Time to copy checkpoints to spare node
γ	Performance loss factor using remaining resources

5.3.3 Spare Node analytical evaluation

To evaluate the proposed model using an analytic approach, a series of example values are developed for an application execution with FT protection. Table 5.3, shows the parameter's values used to obtain the s point, using the model defined by the Eq. 5.14.

When applying the values defined in Table 5.3, to the s value model 5.14, we are able to calculate the starting point where running with remaining resources after a failure is better than setting up a spare node. For this particular case, $s = 0.97$, which multiplied by the ET_{nft} gives a value of 97% of the application execution.

Figure 5.9, illustrates that if a failure occurs at the beginning of the execution, the remaining execution time using only the remaining resources (RET_{ft-s}) is considerably higher compared to resume the application using a spare node (RET_{ft-r}).

When failures occur in the final portion of the application execution, the remaining execution time using the remaining resources, performs equally or even faster than

Table 5.3 Analytic Parameters Values.

Variable	Value
m	0.4
γ	1.3
l	0.5
ET_{nft}	5000 s
t_{rr}	30 s
t_{cs}	150 s
t_{rs}	20 s
σ	500 s

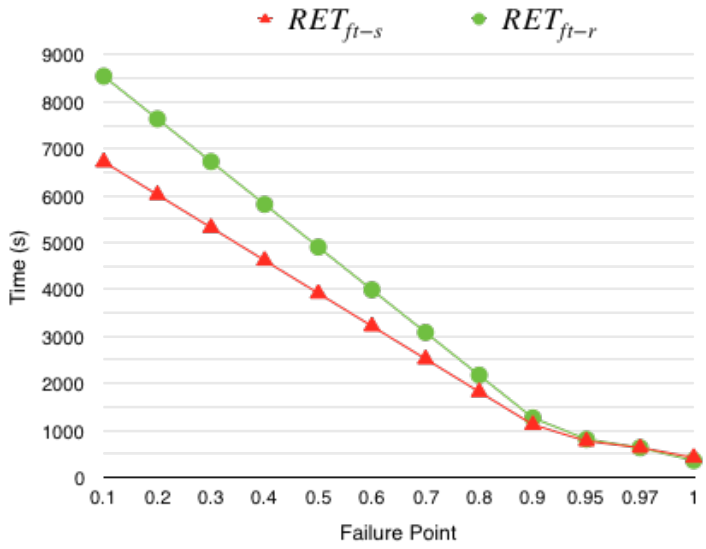


Fig. 5.9 SNCM Analytic Model Evaluation

setting up a spare node and resume the execution. Furthermore, it also helps to reduce the cost function by not using a spare node.

5.4 Experimental Results

In this section, a series of experiments are presented to validate both, FPP and SNCM models. Moreover, the experiments show the overhead reduction obtained applying both models with their corresponding methodologies. The experiments were developed in a controlled environment and injecting failures.

In order to validate the proposed models, a characterization process is done. The values obtained during the characterization are used to calculate the results of the FPP and SNCM models. Both models help determining protection and recovery insights for the application FT configuration. Particularly, the FPP model and, after its k point is calculated, determines where to start FT protection in an effective way. Meanwhile, with the SNCM model, a s value is calculated, which specifies, the point during the application execution where if a failure occurs, it is convenient to resume the execution without a spare node.

Regarding the SNCM model, further characterization is required. Besides measuring the application execution with FT protection, it also includes the FT storage evaluation of the application protected with FT. This allows verifying if the application can run with FT protection and be recovered in case of failures using remaining resources of the execution environment. Once, the verification is finished, and if it is possible

to run the application without a spare node resource, a performance loss factor (γ) has to be measured. This factor is obtained executing the application without one of the execution nodes and comparing it with the execution time with FT. Using the performance loss factor (γ) and the FT characterized variables, the s value is calculated.

After the characterization is done, the models values are obtained. To validate them, the applications are executed and failures are injected in different points of the application execution. For the FPP model, the failures are injected after the calculated k point. Similarly, for the SNCM model, the failures are injected after the calculated s point. To evaluate both model, isolated experiments were done for each model.

The characterization of the parameters that are needed to apply the methodology requires at least one execution for measure purposes, but for applications that are executed several times, this effort can be amortized. Once the parameters for the model are obtained, they can be used as many times as the application is executed in the same HPC system, due to dependence of measure FT parameters with the execution environment. Furthermore, tools such as PAS2P [79], can help predicting the total execution time of the application and may even give parameters such as checkpoint time (t_c).

For the FPP model, the experiments in both scenarios, are done to show if the execution after failure using FT finishes faster than re-executing from the scratch using no-FT protection. Depending on the point of failure after k point, it is possible to determine how much is saved in terms of execution time after a failure using FT protection. The overhead reduction is obtained by deleting the checkpoints that are below the obtained k value. This experimentation shows the overhead reduction for each case, using the presented approach.

The failure injected after the calculated s value, for the SNCM model validation helps to corroborate if using the remaining resources is enough to offer automatic restart limiting the MTTR, and for some scenarios even better than recovering using a spare node. The experiment shows that depending on the point of failure during the application execution a spare node may not be needed, and in this way obtaining a dynamic spare node configuration which achieves better FT resources management.

5.4.1 Hardware and Software Configuration

The experiments were executed in the cluster shown in Table 5.4, and further described in Appendix A.0.1. Parallel applications used in the experiments were compiled

Table 5.4 Experimental environments.

COMP.	AOCLSB-FT	AOCLSB-L	AMAZON EC2
CPU	2 quad-core Intel(R) Xeon(R) E5430 @ 2.66 GHz	8 AMD Opteron (x8) Processor 6262 HE @ 1.60 GHz	8 (vCPU) Intel Xeon E5-2680v2, 2.8 GHz
RAM Memory	16 GB	252 GB	16 GB RAM
Local Storage	30 GB HDD		2 x 80GB SSD
Centralized Storage	NFS: 715 GB (v3)		NFS: 50 GB (v4) EBS (Elastic Block Storage)

with Open MPI 1.7.1. The MPI implementation was compiled and configured with –**with-platform=optimized** option.

The checkpoint tool used to evaluate the designed model is DMTCP (*Distributed MultiThreaded CheckPointing*) [4]. The 2.4.4 version is setup with timing measure option.

The checkpoint interval for parallel applications is calculated using the Fialho et. al. [34] proposal Eq. 5.15.

$$\sigma = \frac{\sqrt{\phi t_c(2\alpha - t_c - 2\Delta_{tr})}}{\phi} - t_c \tag{5.15}$$

The variables description are the following:

- σ : Checkpoint interval time.
- t_c : The time spent on a checkpoint operation including the local (node) storage time.
- α : The mean time to interrupt (MTTI) for a given system.
- Δ_{tr} : The time spent no processing the message log after a fault. In the experiments and as the coordinated checkpoint is used, this variable value is always 0.
- ϕ : The inter-process dependency factor. For the experiments ϕ is always equals to 1, because the checkpoint/restart of all processes in the parallel application is always performed.

The experiments were done using three of the NAS benchmark suite applications: CG, BT and LU with the Class D workload. A Master-Worker Matrix Multiplication is also used for the experiments, with a matrix size of 5120x5120.

5.4.2 Execution Characterization

In this section, the execution characterization is made for both proposed models: FPP and SNCM. The characterization is the first step to obtain the variables for the models, which are dependent to the application and system environment in which users run their executions.

First Protection Point Model

For the FPP, experiments were done using the coordinated rollback-recovery protocol. FTM was setup using the DMTCP checkpoint facility. As seen in the previous section, checkpoint interval is set by using Fialho et al. [34] model, and the MTTI value used is 720 seconds for experimental reasons.

The l value of the model, is measured when a failure is injected during the application execution. Then, the executed time of the last checkpoint interval is measured. As explained, k value allows to calculate the moment in seconds to start FT protection ($k \cdot ET_{nft}$).

The characterization results are shown in Table 5.5. Three cluster environment were used, along with three NAS applications: CG, BT and LU. The CG application ran 8 processes per node in the AOCLSB-FT cluster. In the AOCLSB-L Cluster three benchmark with the following configuration were ran: CG and LU, both using Class D with 32 processes per node, and BT - Class D runs 32 processes in 3 nodes and 25 processes in 1 node. For the cloud environment, 8 c3.2xlarge nodes were setup to run CG - Class D using 8 processes per node. The results of the application of the FPP model are also shown in Table 5.5. It is possible to observe that the overhead of protecting the applications with FTM is in the range of 25% to 40%. The number of

Table 5.5 Characterization and results for the FPP model.

Variables	AMAZON EC2	AOCLSB-FT		AOCLSB-L		(M)easured / (C)alculated
Application	CG-Class D	CG-Class D	CG-Class D	BT-Class D	LU-Class D	-
Processes	64	64	128	121	128	-
Iterations	500	100	500	500	1000	-
ET_{nft} (s)	1792.66	885.29	2035.74	1522.92	1854.21	M
ET_{ft} (s)	2596.51	1101.85	2491.68	2179.87	2355.66	M
t_c (s)	120.70	54.32	75	125	63	M
σ (s)	278	220	245	280	232	C
n	7	8	6	8	7	M
t_r (s)	110	5.91	7	8.24	4.79	M
l	0.51	0.57	0.51	0.50	0.50	M
k	0.50	0.33	0.33	0.45	0.30	C
Start FT at (s)	896.33	292.14	671.79	685.31	556.26	C

total checkpoints performed during an application execution are denoted by n. Between 7 to 8 checkpoints are taken during the application execution.

The calculated k values were used to perform the experiments injecting failures. The goal is to get total execution times (with and without FT) including failure and execution times after failure, to evaluate the effectiveness of the FPP model.

Spare Node Configuration Model

In order to apply the SNCM model, an evaluation of the available resources have to be done. A characterization of the parallel application executed in the described environment is performed. The characterization includes a memory and storage evaluation. This allows verifying if the application can run with FT protection and be recovered in case of failures using the remaining resources.

Experiments were carried out using BT-Class C NAS Parallel Benchmark and Matrix Multiplication Master-Worker Application with static workload distribution. The Matrix Multiplication application is used due to the simplicity on modifying the application workload.

The evaluation begins with the main memory usage monitoring of the application and FT processes. The goal is to determine if there is enough memory to run the processes of the neighbor node, which would experienced a failure. The monitoring procedure verifies the local and neighbor memory usages that allows calculating the free available memory in case of failures.

Figure 5.10 illustrates the memory usage on a node in the execution environment during the execution of Matrix Multiplication application. The measurement is done using *ps* and *free* performance tools. Approximately, 4000 MB is reserved for the

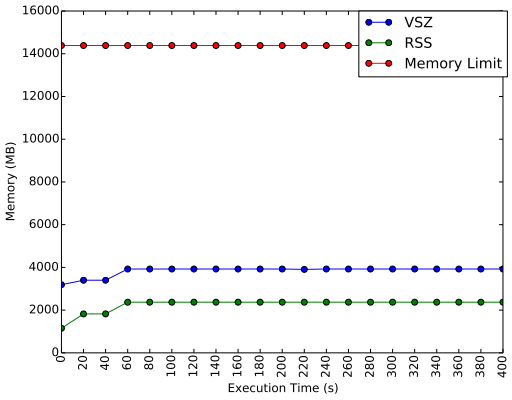


Fig. 5.10 SNCM: Main Memory Evaluation results when no spare is required.

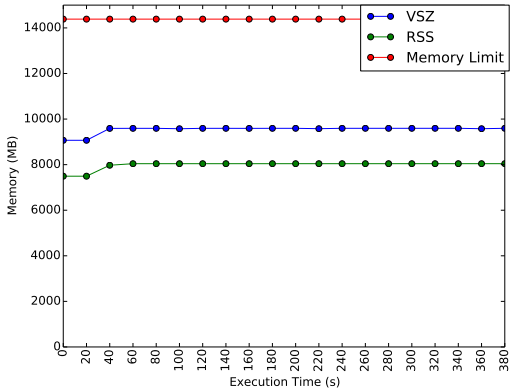


Fig. 5.11 SNCM: Main Memory Evaluation results when a spare is required.

virtual memory (VSZ) that application processes can use, and 2200 MB of the physical memory allocated at the moment (RSS) in which the application is running. It is observed that almost 10000 MB remains free. In this scenario, it is possible to state that the computing node can stand to run its neighbor processes in case of failures. For the experiments, local disks on the nodes, are used to store the checkpoints, and the replication is done to distribute the checkpoint files among the neighbor nodes with FTM.

To illustrate a scenario in which there is not enough memory available to run neighbor processes in case of failures, the Fig. 5.11 shows the memory usage of Matrix Multiplication with a higher workload. It is possible to notice that a spare node is required for this scenario.

FT storage evaluation is made to verify if there is enough available space to save local and neighbor node processes checkpoint files. Figure 5.12, shows the storage

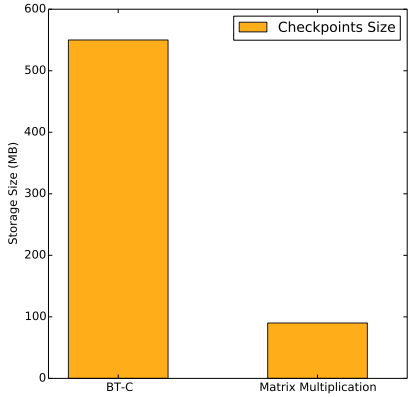


Fig. 5.12 SNCM: FT Storage evaluation.

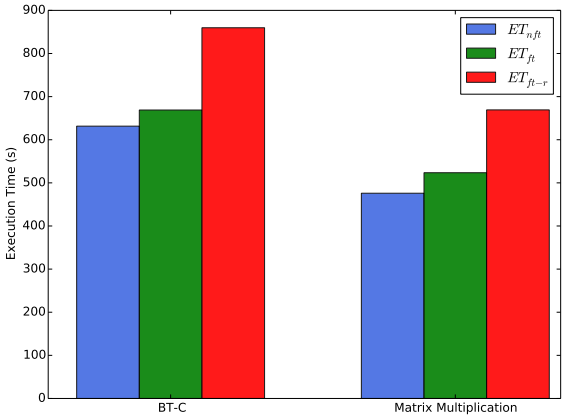


Fig. 5.13 SNCM: Application executions characterization.

usage for the checkpoint files of local and neighbor processes. As the AOCLSB-FT provides 30GB of local disk space, this factor does not represent an issue.

The presented experiments show that an evaluation of memory an storage usage is necessary to determine if the remaining nodes can be used to resume the application after failures. It is essential to know the available resources in order to recover the application's execution after a failure appears. Furthermore, this evaluation determines if the proposed Spare Node model can be applied.

When the application can run without configuring spare node, the SNCM allows determining the impact of losing one compute node. Characterization executions configuring the application with and without FT, and with FT and reduced resources are made. The executions are shown in Fig. 5.13. Reduced resources execution is configured without one node of the cluster. The latter is done to measure the performance loss factor (γ).

Using the parameters obtained in the characterization, the point s can be calculated by applying the model Eq. 5.14. The point during the execution, when the recovery from a failure and continue the execution without using spare nodes is calculated by multiplying $s \cdot ET_{nft}$, and it is named "No-Spare Point" in the Table 5.6, which shows the parameters to calculate the s point with the Spare Node model.

Table 5.6 Characterization and results for the SNCM model.

Variable	BT-Class C	Matrix Multiplication	(M)easured / (C)alculated
App. Processes	64	64	-
App. Iterations/Matrix Size	100	5120x5120	-
m	0.06	0.10	C
l	0.5	0.1	M
γ	1.28	1.27	C
s	0.9776	0.9988	C
No-Spare Point (s)	617.40	475.42	C
ET_{nft} (s)	631.51	475.96	M
ET_{ft} (s)	668.92	523.49	M
t_c (s)	8.97	5.50	M
σ (s)	85.34	67.79	C
t_{rr} (s)	2.22	3.33	M
t_{rs} (s)	2.00	2.91	M
t_{cs} (s)	16.65	2.47	M

5.4.3 First Protection Point Evaluation

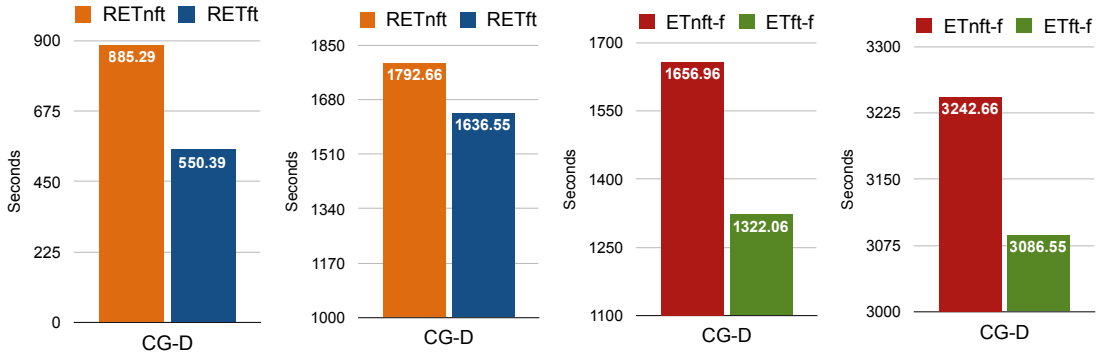
In the experiments, applications were executed and failure injection is made after the executions overpass the k point, in order to verify that in terms of total execution time is better to protect the application with FT, against re-executing it from the beginning. The objective of this experimental scenario is to evaluate and validate the FPP model. Table 5.7, shows the failure injection point during execution on each application in the corresponding cluster.

When the failure is injected inmediately after the k point, two time measurements are made: the remaining execution time without FT (RET_{nft}) and the remaining execution time with FT (RET_{ft}) are measured. Both measurements, are made after the application experienced the failure. Regarding the RET_{nft}, no of repair time ($MTTR = 0$) was considered.

In Fig. 5.14a for the AOCLSB-FT Cluster, it is possible to observe the remaining execution time RET_{nft}, represents the time of restarting the application after the failure is detected, for this scenario, the failure is injected after 771.67 seconds of execution, and results show the RET_{ft} reduced 38% the RET_{nft}, which means a reduction of approximately 335 seconds. Similar results are shown for AMAZON EC2 and AOCLSB-L execution environments in Fig. 5.14b and Fig. 5.15a.

Table 5.7 Failure point for applications in the execution environments after the k point

Application	Cluster	Failure Point (s)
CG-Class D	AOCLSB-FT	771.67
	AOCLSB-L	1033.52
	AMAZON EC2	1450.00
BT-Class D	AOCLSB-L	1210.31
LU-Class D	AOCLSB-L	1318.06



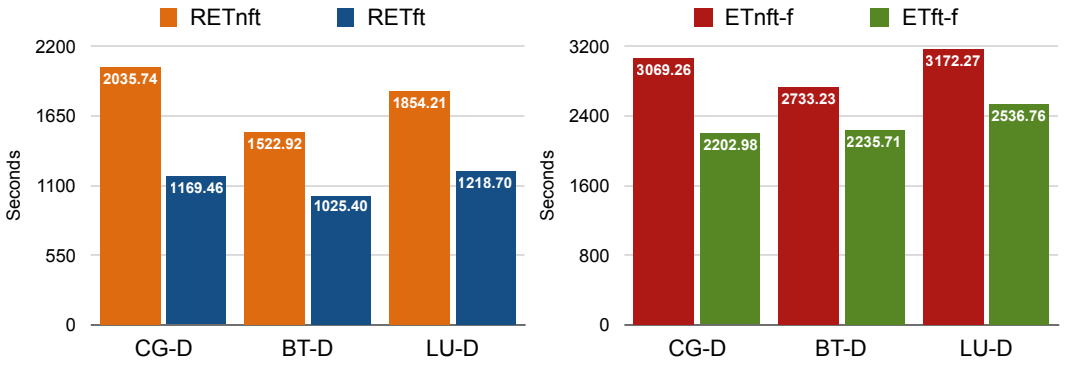
(a) AOCLSB-FT (b) Amazon EC2 (c) AOCLSB-FT (d) Amazon EC2

Fig. 5.14 Remaining and total execution time with FT and without FT for the AOCLSB-FT cluster and Amazon EC2 cloud.

The Fig. 5.14c, 5.14d and 5.15b, effectively show that protecting the application with FT give a more effective total execution time ($ET_{ft-f} < ET_{nft-f}$), when a failure appears immediately after the k point. For instance, executing CG-Class D application in the AOCLSB-L cluster using FTM, experiencing a failure immediately after the k point, finishes the execution 28% faster than without FT.

The previous set of experiments verifies the functionality of FPP model, which determines a k value for the application that defines the starting point where to insert FT protection. It is possible to state that checkpoints below k value can be avoided.

This proposal proposes taking out checkpoints below the k value, though maintaining checkpoint interval. The natural consequence of removing the checkpoints is the reduction of FT protection overhead. In practical terms, avoiding the checkpoints



(a) Remaining Execution Time (b) Total Execution Time

Fig. 5.15 Executions with and without FT of CG, BT and LU applications for the AOCLSB-L cluster.

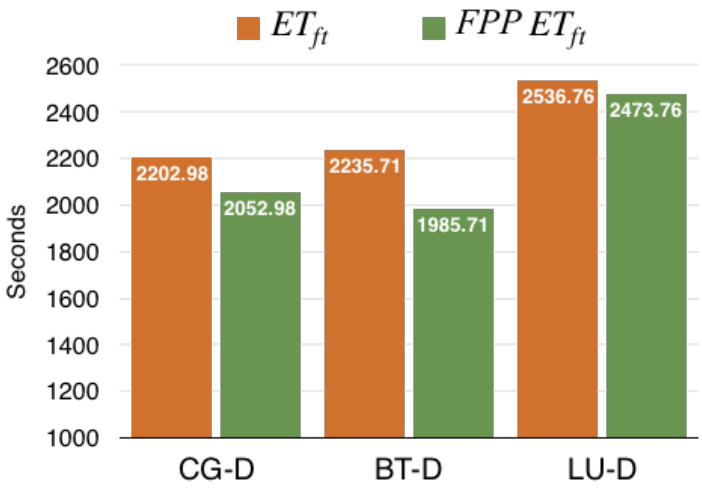


Fig. 5.16 Overhead Reduction in AOCLSB-L Cluster using FPP.

below the k point, gives an overhead reduction of approximately 7% (CG-D), 11% (BT-D) and 2% (LU-D) shown in Fig. 5.16.

5.4.4 Spare Node Configuration Model Evaluation

After the initial characterization is done, the evaluation of running the application without spare node, using only the remaining resources is used to compare the performance loss against the costs of setting up a spare node. The costs can be, transfer of checkpoint files, with a slow network, and even the provision, which for cloud environments could take time, and for bare-metal the provision may not be available.

In the experiments, applications were executed and failure injection was made immediately after the executions reach the s point, in order to verify the SNCM. Failure injection point for NAS BT-Class C is at 643.16 seconds of the application execution and at 499.45 seconds for Matrix Multiplication. For both scenarios after the calculated s point ($s \cdot ET_{nft}$), in order to confirm that after the execution reaches the point s, it is not mandatory to setup a spare node to continue the execution and get the expected execution results.

Figure 5.17 depicts the execution after failure of the both applications using a spare and reduced resources after the injected failures. Also, it is possible to observe different factors that have influence on the final execution time, such as the time to copy checkpoints to the spare node (t_{cs}) and the performance loss factor (γ). It is possible to notice that depending on the spare node availability and network performance for protection information translation, sometimes its better to continue the execution with

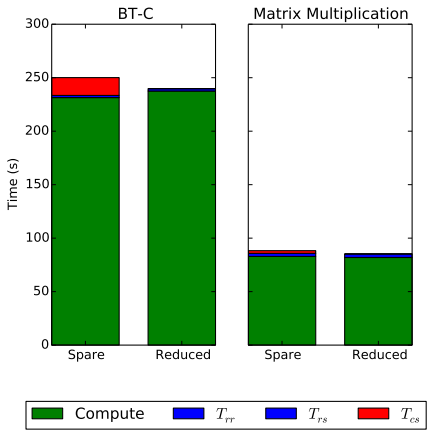


Fig. 5.17 SNCM Evaluation: Recovery factors in the execution after failure for the BT-Class C and Matrix Multiplication applications.

the remaining resources and pay the performance loss factor, which can slow down the execution, but still it can finish execution faster than with a spare node.

The Fig. 5.18 shows the total execution time, including failures with spare (ET_{ft-fs}) and with the remaining resources (ET_{ft-fr}). For instance, the BT-C application, resumes its execution after failure approximately 11 seconds faster than using a spare.

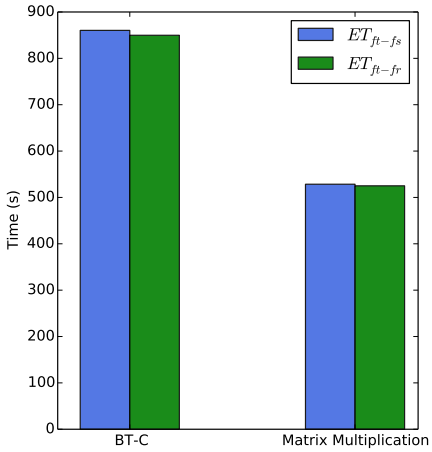


Fig. 5.18 SNCM Evaluation: Total Execution Time.

5.5 Summary

This chapter presents two models that can be used to configure the protection and the recovery of FTM. Although, the models can be used to configure other fault tolerance frameworks that implement rollback-recovery protocols. The models are designed in order to improve the FT resources configuration for the users, applications and system environment specific requirements, always maintaining an acceptable level of FT protection.

The First Protection Point model aims to reduce the overhead by avoiding checkpoints during the application executions, by calculating an starting point to start the FT protection. FPP model allows to increase the application performance protected with FT up to 11% for the presented experimental scenarios.

Spare configuration represents an important issue for the FT protection. It can determine if an application can continue its execution despite failure appearance. The presented chapter shows a method to evaluate and determine if an application can be executed and protected with FT without spare node. Furthermore, presented experiments allow concluding that in certain failure scenarios a spare node is not a mandatory resource, and as shown in the experimentation, avoiding a spare node, for some scenarios, is more convenient than to setup it. The SNCM model successfully identifies a point during the application execution, after which, in case of failure, setting up a spare node can be avoided, obtaining a more suitable spare node configuration, which reduces the resources cost.

Chapter 6

RaaS: Resilience as a Service for HPC in Cloud Environments

"To get through the hardest journey we need take only one step at a time, but we must keep on stepping."

Chinese Proverb

Moving HPC application's executions to cloud environment is challenging. Large cloud environments are prone to failures, putting at risk the user's application execution. In this chapter, Resilience as a Service (RaaS) is described, which provides a fault tolerant service on cloud environments. The architecture overcomes the challenges of bringing traditional FT architectures to a dynamic execution environment, such as cloud ecosystems.

Traditionally, HPC applications executing on bare-metal clusters rely on fault tolerance (FT) solutions that are designed to operate with a static number or sometimes limited number of physical resources. Cloud environments offer an unique opportunity compared to traditional cluster FT, which are the seamless infinite resources available in cloud, and its ability to create and destroy virtual machines immediately in just few seconds.

RaaS takes advantage of the flexibility offered in cloud environments to provide virtualized resources dynamically. It is able to abstract the underlying resources that are being used for FT protection tasks, making the design applicable to multiple cloud providers (e.g. OpenStack, Amazon AWS, GCE, Azure, Digital Ocean, etc), and to different storage services that are commonly provided on cloud environments. The RaaS design makes it possible to integrate new cloud services that can be used to

provide FT, independently of a specific cloud provider. The outcome is a FT service designed for cloud environments that provides multiple configuration possibilities to fulfill diverse users' requirements.

In this chapter, the RaaS modules are detailed, along with the procedures to protect HPC applications. The FTM framework (Chapter 3) is used inside the RaaS architecture to offer high availability, with a solution that is distributed, scalable, automatic and transparent. The proposal aims to abstract users from the FT-specific complexities.

The rest of this chapter is organized as follows: section 6.1 elaborates an overview of the proposed solution. In sections 6.1.1 and 6.1.2 the Resilience as a Service (RaaS) and its main components are further detailed. Section 6.2 explains the experimental validation design, the applications used on the evaluations and the performance results. Lastly, a discussion is made about RaaS limitations in section 6.3.

6.1 Proposal Description

Fault tolerance is often a requirement in HPC to provide applications with the ability to continue the execution in case of system faults. This requirement is also essential in clouds given the probability of faults and also because the application owners are relieved of the management of the underlying infrastructure which is the responsibility of cloud operators. This work introduces Resilience as a Service (RaaS), which provides a cloud-native FT solution for MPI applications that run on virtual clusters in the cloud platform.

The solution borrows the distribution, automatic and transparency concepts of FTM (Chapter 3) to provide high availability as a service to multiple applications running on virtual clusters (VC) in cloud ecosystems. Furthermore, RaaS supports on-demand resource allocation for FT recovery purposes, making use of the elasticity characteristic offered by the cloud. In addition, it provides the ability to enable multiple storage configuration types for FT protection tasks, due to its abstract definition of FT storage.

The user's application register their MPI parallel applications on the RaaS service, prior to execution. The RaaS service configures the components needed to provide FT to the application execution, such as the protection storage devices and the node monitoring facility. When a failure occurs the recovery procedure is initiated using the protection data to recover the application execution and relying on the cloud-intrinsic on-demand allocation of resources when necessary. The application resumes

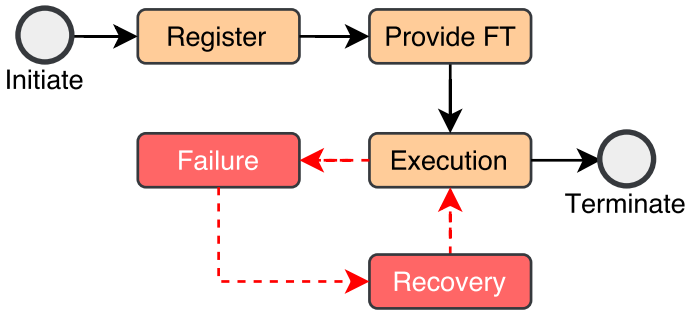


Fig. 6.1 RaaS service general functionality procedure.

its execution after the recovery is finished. The basic working procedure is shown in Fig. 6.1.

RaaS is built with three main components: the RaaS Web Service, which allows users to register their applications to enable FT protection. The Fault Tolerance Manager Daemon (FTMD) which manages and distributes the FT protection information, and it is also responsible for updating the alive information of instances in the virtual cluster along with the RaaS Web Service. The component functionalities are similar to the Protector controller of the FTM solution. The third component is the Cloud Manager (CM), which makes possible the interaction between RaaS and the cloud provider. This interaction is necessary to enable the dynamic provisioning of spare node instances and for configuring resources to perform FT-related tasks.

The main components of the RaaS service and their interaction are depicted in Fig. 6.2. The proposed solution and its functions are transparent to the user's applications, following FTM transparency concept.

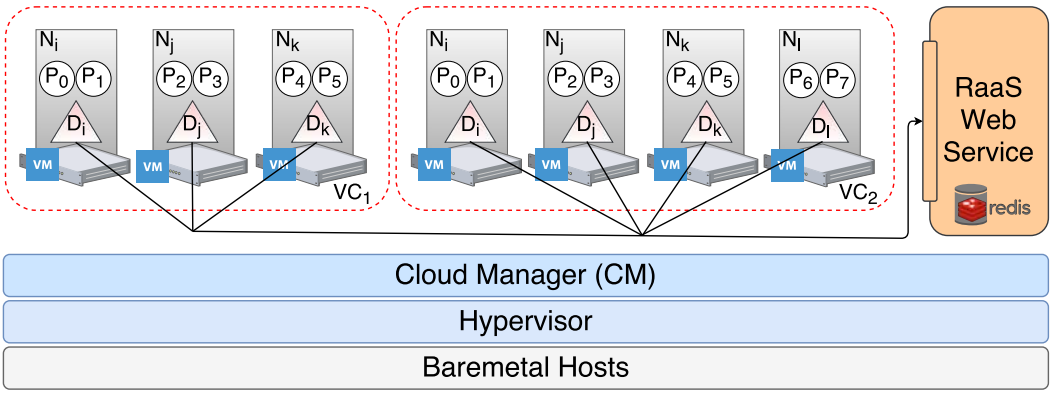


Fig. 6.2 RaaS general architecture: Each node of the virtual cluster has a FTMD (D) that communicates with the RaaS Web Service.

6.1.1 RaaS Web Service and Cloud Manager

The RaaS Web Service provides functionality for registering the user's applications and enables FT protection as a feature of the cloud environment. The functionality is implemented in four managers and exposed through a RESTful API, as depicted in Fig. 6.3.

The web service is in charge of the configuration and installation of the FT components on the virtual clusters where the user's application will be executed. It also maintains the information about the state of each instance of the virtual cluster where the user's application is running.

Actions are triggered automatically from RaaS when failures are detected in order to recover the application execution. The procedure is completely transparent to the users. In addition, the web service is also responsible for the interaction with the Cloud Manager (CM), to dynamically acquire resources for the recovery when necessary.

The application's registration and status, virtual clusters, and related information is stored in a database. The Redis[1] in-memory database is used, for this purpose. Redis is a key-value database that was selected to improve the access performance to the information of the user's application from the web service.

The RaaS Web Service is RESTful and it is designed using the OpenAPI[2] specification. It works along with a Cloud Manager (CM), which in this paper is provided by OpenStack as an use case. However, the API is able to work with other CM implemen-

[1]https://redis.io
[2]https://www.openapis.org

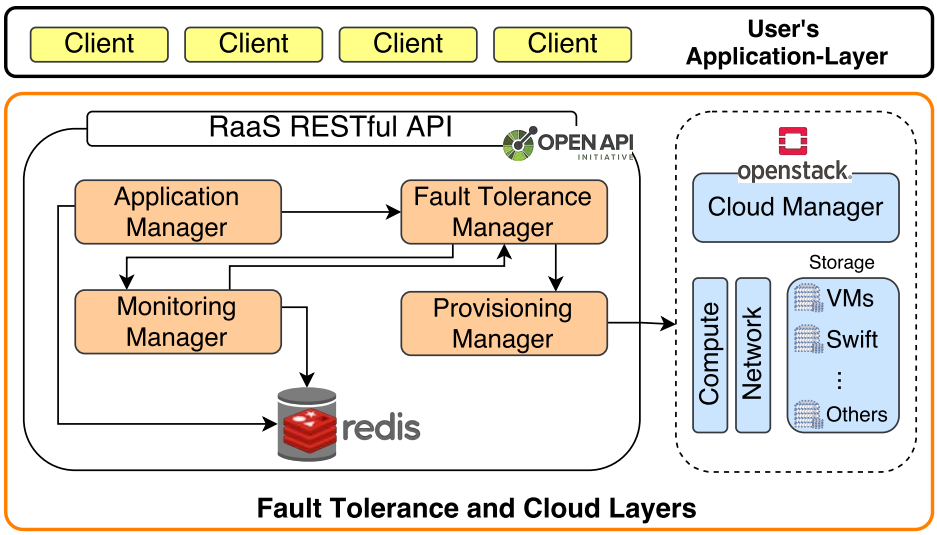


Fig. 6.3 RaaS Web Service components.

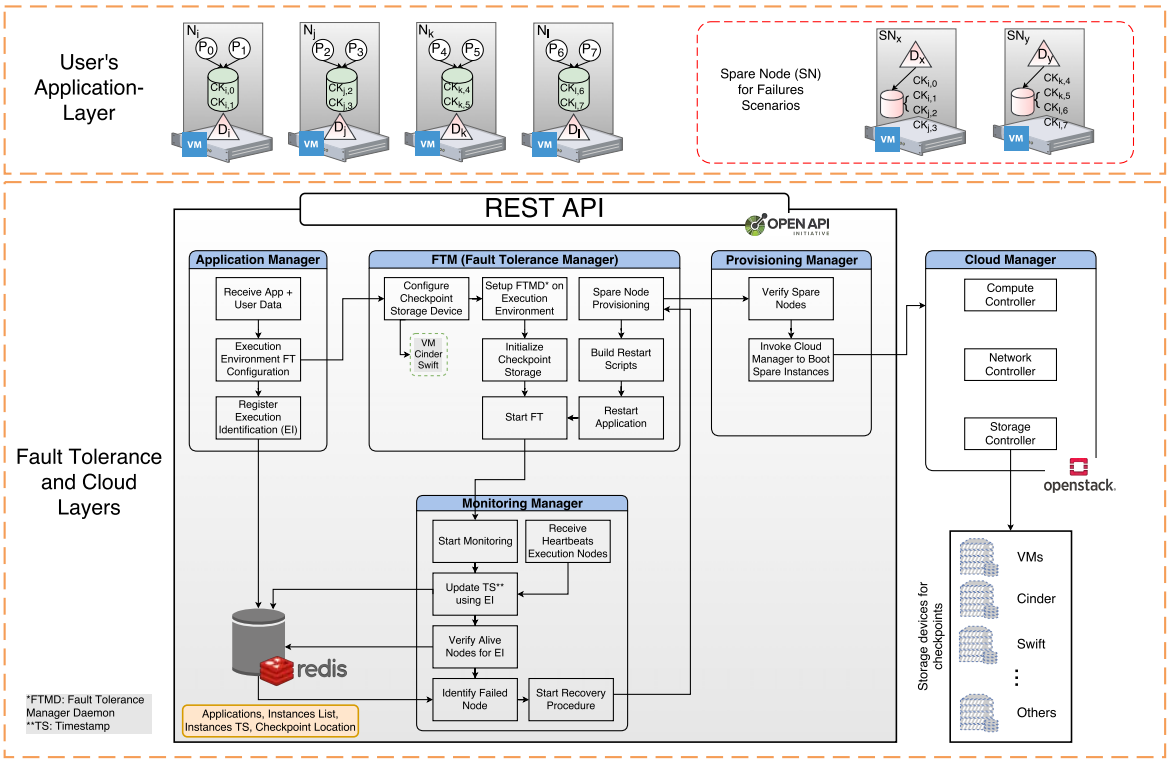


Fig. 6.4 RaaS: Components interaction.

tations of different cloud providers, due to the modularity of our software design. The web service is implemented and published using Python Flask[3]. A complete schema of the RaaS components is shown in Fig. 6.4. The figure shows an Application Manager to register applications and environment; a Fault Tolerance Manager enabling FT services for protection and recovery; a Monitoring Manager for failure detection; and a Provisioning Manager for dynamic provisioning of resources.

Application Manager

This module registers the user's application that is going to be executed in a virtual cluster on the cloud environment. The registration process requires i) the user identification, ii) the virtual cluster identification, iii) the execution and spare node instances list, iv) the MPI application binary path, and v) the quantity of processes. This information is stored in the Redis database, and it is used by the RaaS to identify each user client executing applications in the various virtual clusters.

The registration process also requires information regarding the type and amount of resources that the user wants to dedicate for fault tolerance. Currently, the RaaS

[3]http://flask.pocoo.org

service is capable of supporting block, object, file and VMs for storage purposes – however, other persistent or ephemeral options can be easily added following the same software design. For each storage type, users have to define the location and path to such resources in order to be accessed by RaaS.

Once the information is completed, the web service will process the request of registration. During this process, the registration information is stored in Redis using as key the calculated hash value of all the parameters used in the request The value associated to this key is a the JSON[4] structure containing the registration data. This allows to uniquely identify the user, the application and execution parameters.

After the registration process, the RaaS service configures the FT mechanisms and the application checkpoint time. This information is used together with the *mean time to interrupt* (MTTI), which is dependent on each platform. The FT setup is invoked by the Application Manager, although it is executed by the Fault Tolerance Manager. Another task, is to store the current time-stamp (TS) in the database for each instance of the execution environment. TS is used to detect node failures at runtime with the Monitoring Manager. This allows users to execute application with fault tolerance protection.

Fault Tolerance Manager

The Fault Tolerance Manager configures and installs FT in the user's execution environment for their applications. Furthermore, this manager is responsible for the application recovery in case of failures, using available spare nodes and/or creating more instances when required.

The fault protection of applications stands on top of the *coordinated rollback-recovery protocol* [30] which orchestrates all application processes to take checkpoints and create a recovery line [10]. The checkpoints are taken using DMTCP [4], which acts as the observer controller in the underlying RADIC architecture.

There is one checkpoint file per application process. RaaS uses Daly's model [22] to obtain the optimal checkpoint interval (σ).

$$\sigma = \sqrt{2\alpha t_c} - t_c \tag{6.1}$$

The model (Equation 6.1) uses MTTI of the execution platform (α) and the checkpoint time of the application (t_c), which requires the characterization of the application.

[4]http://www.json.org

During the FT installation and configuration of the execution environment, the manager sets properties to the Fault Tolerance Manager Daemon (FTMD) that is going to be installed and launched on each instance of the runtime. The properties contains information about the location and the type of device to store the checkpoint files of each process on each execution instance. The checkpoint interval is also set as a property for the FTMD. After all properties are set, the FTMD is packaged and sent to each node instance for installation. After the installation has finished, the FTMD is launched on each node instance of the execution environment.

The initialization of the checkpoint storage depends on the parameters that are obtained from the Application Manager during the application registration. When users chose to store checkpoints in a block storage, the information about the location is configured in the FTMD on each node instance of the execution environment. RaaS also provides the option to store the checkpoint files in VMs, which also works as spare node instances. For the latter, the checkpoints are first stored into local ephemeral disks of each node instance of the execution environment and, then, are distributed to the spare VMs. After occurrence of a failure, the VMs are used as a replacement for the faulty node instances, on the virtual cluster, to resume the execution after the recovery procedure.

In case of a failure, the Monitoring Manager invokes the recovery procedure which is executed by the Fault Tolerance Manager component. The manager identifies the location of the checkpoint files, and the availability of the spare nodes. If there are enough spare node instances, no provisioning is needed, otherwise spare nodes instances are created, provisioned and registered before resuming the application. In case the spare nodes are newly created, the checkpoints are transferred. Applications are resumed using a script, which uses the checkpoint files, on the new execution environment. Also, FTMDs are launched on the instances, accordingly.

Monitoring Manager

This module is responsible for detecting runtime failures at node instances. It constantly monitors the status of each node in the virtual cluster executing the application. The monitoring is initiated when the application is registered. At this point, a time-stamp (TS) is stored in the Redis database for each node on the virtual cluster of the execution environment. Once the FTMD is launched on each node instance of such virtual cluster, it starts sending a constant heartbeat message to the RaaS Web Service. The Monitoring Manager processes each heartbeat message and update the Redis database with the current TS.

A timeout verification is performed in the RaaS service, of the virtual cluster nodes by checking the database last stored TS. This is an important task of the FT solution as it allows for detecting faults whenever any of the node instances does not respond within the timeout limit. A detection of a fault also invokes verification on the remaining instances in the virtual cluster. This allows to detect multiple failures such as bare-metal host faults in the platform. Finally, the recovery procedure is triggered and performed in conjunction with the Fault Tolerance Manager.

Provisioning Manager

The main function of the Provisioning Manager is to interact with the Cloud Manager (CM) to acquire resources in case of failures when needed. In the case of a failure, the Fault Tolerance Manager component invokes the Provisioning Manager component to update and configure the execution environment. The Provisioning Manager component receives information of the quantity of instances that have failed, and verifies the availability of initially configured spare nodes. If there are not enough spare nodes, it requests to the Cloud Manager component the number of instances required to resume the execution without performance degradation. This is a blocking process, meaning that the Provisioning Manager component waits until the request is fulfilled by the CM, i.e. all the necessary virtual resources are instantiated and provisioned.

6.1.2 Fault Tolerance Manager Daemon, FTMD

This is a stateless daemon configured and installed on each node of the virtual cluster wherein the user's application is going to run. It handles the distribution of the local checkpoint files to the storage location set by the Fault Tolerance Manager. Heartbeat messages are sent to the RaaS Web Service to notify the alive information of each node.

FTMD works as a daemon, and it uses two main threads. The first thread is used to send the heartbeat information periodically to the RaaS Web Service. This heartbeat is used on the Monitoring Manager to detect failures as described above.

The second thread is only activated when the checkpoints are stored in VMs that are used as spare nodes. In this schema, the execution nodes first store the checkpoints locally in ephemeral disks for better performance and, then, this thread distribute them to the location configured in the Fault Tolerance Manager. The distribution of the checkpoint files is accomplished by following a round-robin technique.

6.2 Experimental Validation

In this section, a series of experiments are presented to evaluate RaaS, using Nbody simulation, a real application, and a NAS benchmark in a OpenStack private cloud environment built on top of a cluster with configurations defined in Table 6.1, further details are described in the Appendix A.0.1.

The experiments are aimed at assessing performance and cost of the RaaS service when applied to various applications using different FT configurations. Faults are injected to show how the system performs the recovery and let the applications end successfully. Moreover, the evaluation shows insights of the cost and performance for the application execution in one or multiple VMs and hosts failure scenarios.

A controlled environment was used to perform the experiments with faults injection. The evaluation includes failures at multiple levels, such as virtual and physical hosts faults. One failure to emulate a virtual instance fault and multiple failures for physical host faults. Each evaluation was executed up to 35-50 times. For measuring checkpointing and restart operations, the built in time measurement tool of DMTCP was used. Other time measurements are done with ***time*** system tool.

Experiments were ran on instances launched with the OpenStack software suite. The version of the different OpenStack projects used to build the cloud environment are shown in Table 6.2.

For the evaluations, different implementations of parallel applications were selected to validate our the proposal:

- **NAS-CG**: This is the Conjugate Gradient method implementation included in the NAS benchmark suite. The experiments are performed using Class D, which has 1500000 rows and 100 iterations.

Table 6.1 OpenStack Cluster (GORO).

Component	Details
CPU	7x Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz
Cores	48 (w/ HyperThreading)
HD	1080 GB
RAM	125 GB
Network Interface	Up to 1000 Mbps (full duplex)

Table 6.2 OpenStack suite installation

Project	Version
Nova	14.0.6
Neutron	6.0.0
Glance	2.5.0
Heat	1.5.1

- **Nbody Simulation**: This is an implementation of the Nbody particle simulation [1], using a circular pipeline parallel computation model with a SPMD technique. The simulation is configured to run with 800000 particles and 10 iterations.

6.2.1 Performance and Cost Evaluation

In this section, we use the class D NAS CG benchmark with 16 processes, and Nbody with 32 processes to evaluate performance. Both applications are executed over 16 instances in the virtual cluster, and are protected using coordinated checkpoints. The optimal checkpoint interval is calculated using Daly's [22] model (6.1). The checkpointing utility used for this work is DMTCP [4], which is a distributed multithreaded checkpointing tool that allows to take application checkpoints transparently. The instance flavor for the experiment is m1.medium, described in Table 6.3. The instances are distributed among all the bare-metal hosts in our environment.

Experiments are performed with different FT configurations, supported by RaaS. The idea is to show the performance and cost of each configuration. Furthermore, executions are done with and without failures, to give users a tradeoff to configure the FT in order to get high availability in a cloud environment.

The performance is evaluated measuring the execution time of the application execution for each scenario, and the cost takes into account the amount of time that

Table 6.3 Flavors

Flavor	VCPUs	Disk (in GB)	RAM (in MB)
m1.medium	2	40	4096
m1.large	4	80	8192
m1.xlarge	8	160	16384

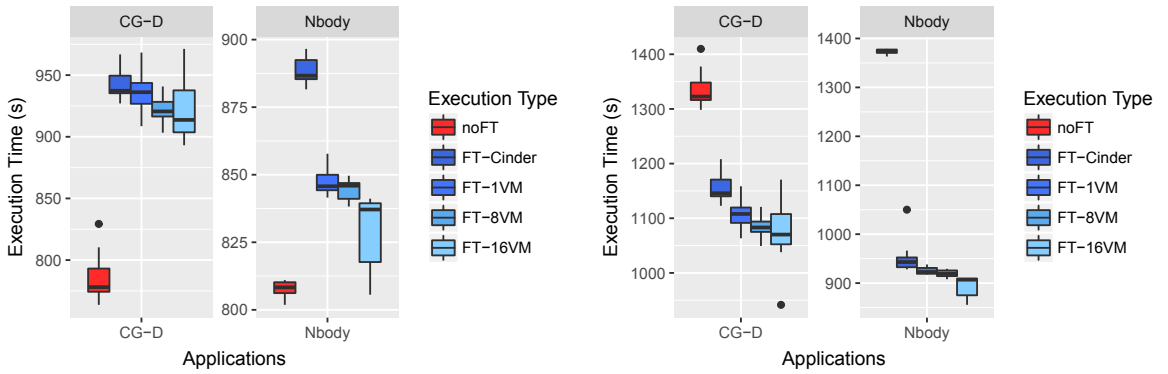
the resources are used in a cloud environment. The experimental scenarios in which the applications were ran, are defined as follows:

- Application executed without FT (**noFT**): In this scenario, the application is run without fault tolerance protection. When failures appear, the application has to be executed from the beginning, losing all the data previously computed. In this case, provisioning compute nodes is accomplished manually and there is no failure detection mechanism in place.
- Application executed with FT using a block storage volume (**FT-Cinder**): The applications are protected by the RaaS service using a Cinder volume, provided by the cloud infrastructure. Cinder volumes are mounted on a NFS server, which serves the volume to the node instances. The application checkpoint files are stored directly into the Cinder volume through the NFS server. The local path to the NFS filesystem is set during the application registration. No spare nodes are configured in this scenario; hence, when a failure occurs, new instances are provisioned by RaaS.
- Application executed with FT using VMs for checkpoint storage and spare node instances (**FT-VM**): In this scenario, the applications are registered to RaaS to obtain FT protection, and users can select an arbitrary amount of VMs. We show results for three configuration alternatives with 1, 8 and 16 VMs to store checkpoints and, at the same time, use them as spare node instances. The checkpoints are initially stored in the local ephemeral disk of each node instance during the execution. Then, they are distributed to the spare VMs using the FTMD. We name each alternative as **FT-1VM**, **FT-8VM** and **FT-16VM**.

To calculate the optimal checkpoint time (σ), using Daly's [22] model (Eq. 6.1), each application was characterized on the execution environment. For the experiments, the MTTI value of ($\alpha = 1000$) seconds is considered. Table 6.4, shows the values obtained by characterizing the applications in the execution environment.

Table 6.4 Applications characterization

Application	t_c	σ	Total checkpoint size (MB)
NAS CG	45.9	257	14000
Nbody	8.0	76	384



(a) Without failures.

(b) With 1 instance failure at 70% of the application execution.

Fig. 6.5 Performance evaluation.

As previously mentioned, fault free and faulty executions are shown. Faulty executions have one and multiple node failures. The fault injection is performed at 70% of the application execution. After the failure, the execution rolls back to the last checkpoint, and the execution is resumed.

The application overhead is calculated for the available configuration alternatives, the results also illustrate the cost of each possible configuration. Figure 6.5a shows the overhead added to both applications with different FT configurations when the execution has not failures. The overhead for the NAS CG Class D benchmark is in the range of 16%-20%. It is possible to observe that when using the configurations with VMs as checkpoint storage and spare node instances, the overhead is reduced. This is the result of the distribution of the checkpoints that is performed in parallel with the application execution. The more resources are available to distribute the checkpoints, the better performance. For Nbody application, increasing the number of spare VMs results in an even higher overhead reduction, which is in the range of 2%-10%.

When the execution includes fault injection, and there is not FT protection, all the computation done until the failure is lost and it is necessary to completely restart the application. In Fig. 6.5b, it is possible to observe that with FT, the execution time including the recovery procedure is shorter than without FT. It is important to note that without FT, the recovery must be done manually, and the time to restart the execution depends on human intervention. The experiments assumes the best possible scenario for executions without FT, assuming that the detection and recovery is performed immediately upon a fault occurs.

The evaluation of RaaS for single and multiple instance failures is shown in Fig. 6.6. Specifically, the figure depicts the execution time of the applications affected by

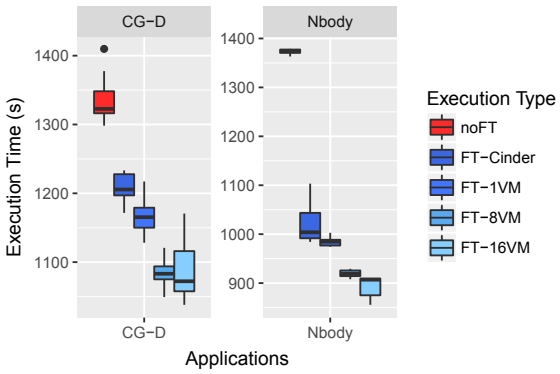
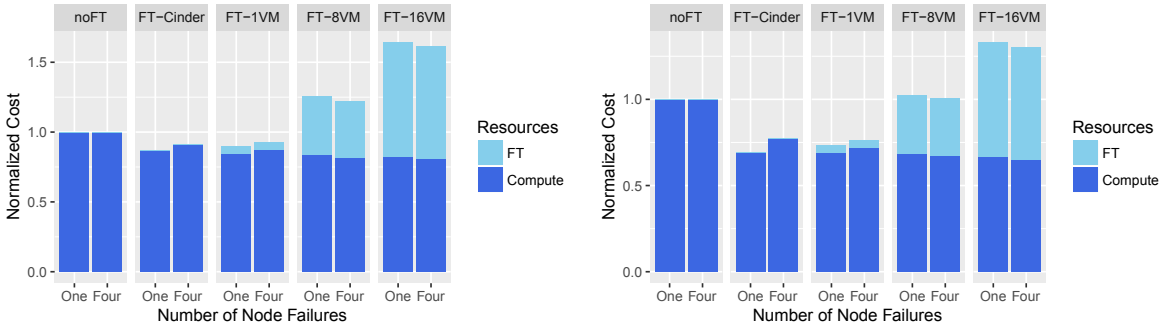


Fig. 6.6 Performance evaluation with 4 simultaneous instance failures.

4 simultaneous instance faults - which allows for emulating a host failure. It can be noticed that the execution time for the FT-Cinder configuration is larger compared to the FT-VM in both failure scenarios i.e., 1 and 4 instance faults. As FT-Cinder has not spare instance initially configured, the recovery time is larger due to the instance provisioning that is necessary to restore the execution. Furthermore, when there are 4 simultaneous instance failures, the configuration with at least 8 VMs shows better performance due to the availability of spare resources upon failures, in this case no provisioning is needed.

Figure 6.7a and Fig. 6.7b show the cost in terms of the time used by each instance for the scenarios with one and four simultaneous faults. It can be observed that the FT-Cinder configuration costs less than the baseline noFT approach with a faster execution when failures occur. For the FT-VMs configurations, the execution performs even faster than FT-Cinder – although the costs is higher as more resources are used.



(a) NAS-CG application

(b) Nbody application

Fig. 6.7 Cost evaluation

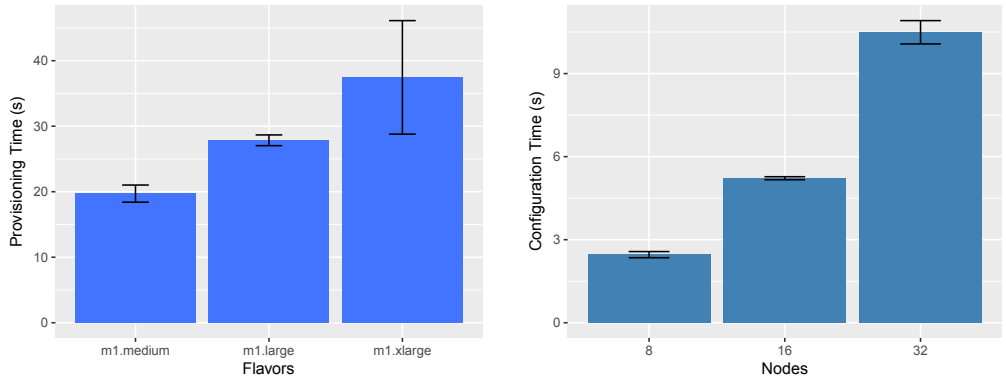
However, when more simultaneous failures occur, and there is enough spare VMs, the final cost is reduced as well as the execution time.

6.2.2 Evaluation of RaaS Operations

The first operation that users do to enable FT using RaaS, is to register the application. Then, the RaaS web service configures the execution instances, and FT resources, to initiate the application execution. The configuration time taken by RaaS is shown in Fig. 6.8b. During the configuration, FTMD is installed and configured in each node of the execution environment. The higher number of nodes executing the application, the longer it takes to configure the FT service.

RaaS detects faults and starts the recovery process. Detection is performed by the web service in conjunction with the fault tolerance manager daemon (FTMD). FTMD run on each instance and sends heartbeats every 10 seconds to the web service which processes them in parallel, using one thread per client. The timeout limit to detect failures is 15 seconds. In the Fig. 6.9, it is possible to note that the detection operation takes similar time to detect failures for a different number of instances, demonstrating the scalability of the RaaS.

The provisioning operation needs to be done inline with the recovery in case of failures for some configurations (e.g., FT-Cinder scenario). The provisioning includes VMs instantiation and initialize the execution from checkpoints. The VMs flavors of the new instances need to be the same than those of the faulty VMs. Figure 6.8a shows the provisioning time for different flavor sizes. As expected, larger flavors, demand longer provisioning time.



(a) Provisioning time. (b) Configuration time.

Fig. 6.8 RaaS operations evaluation.

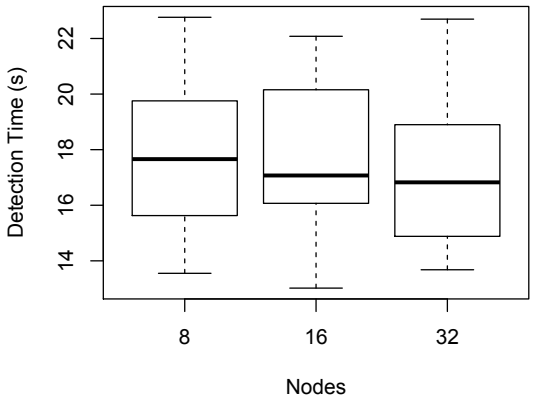


Fig. 6.9 RaaS detection time for node instances.

Larger flavors with more resources (e.g., cores, memory, etc.), requires fewer VMs to run application processes. Although, when using FT-Cinder configuration, failures implicates more recovery time, due to the longer provisioning time. To avoid this issue, users may use the VMs checkpoint storage approach since it also provides for spare nodes, and at the same time, performs faster as shown in section 6.2.1. The larger amount of VMs for FT purposes, the better is the performance and the support for simultaneous failures, although it can be a costly solution.

6.3 Discussion

We have shown that RaaS is efficient and feasible in practice when applied to real scenarios in cloud. However, RaaS is still at an early development stage and requires further research. In this section, we elaborate on its limitations and how we plan to address them.

RaaS offers a FT solution that abstracts the components for protection, detection and recovery and offer them as a service. The software design is extensible and, in theory, supports any cloud provider. Yet, we need to confirm such capability with more relevant cloud providers and platforms (e.g. AWS, VMware, Microsoft Azure, etc) by adding specific implementation of components for these target clouds. RaaS key operations rely on storage performance, provisioning operations and network transfers. When adapting RaaS to several cloud providers, the most cost effective solution would be the one that perform better these key operations.

RaaS leverages cloud capacities like on-demand resource provisioning and diversity of resources. At the same time, it proposes multiple FT configuration alternatives that meet a user defined tradeoff between cost and protection performance.

Despite that, not all potential scenarios are yet addressed. During protection – and for configurations where the VMs are used as storage for checkpoints – there is a period of time where the checkpoint files are distributed to other components. This period is considered critical, as the application has no fault tolerance whatsoever throughout this time because the checkpoint files are unavailable. Although it is possible to use older checkpoint files to restore the application, the wall clock time can be affected as some computation can be lost.

Another limitation is the fact that RaaS components can be affected by a failure as well. The service has several components that work independently such as database (Redis), REST web service API and FTMD daemons. Each of these components are running in virtual machines that can also fail with certain probability. RaaS can overcome failures affecting the FTMD daemons and recover the execution runtime. However, if a failure affects the Redis database or the REST API Web Service, our current implementation is not able to restore the execution of the applications in the virtual clusters. Similarly, the current RaaS service implementation will not recover from failures that affect the VMs where applications are running and, at the same time, disrupt the instances hosting the checkpoints. In this case, checkpoints are not possible to retrieve; a scenario that can happen when a bare-metal host is affected by a failure, making the virtual resources unavailable. One approach we are exploring is to build cloud protection hierarchies, using the same ideas of protection distribution. In this case, RaaS would be protected with the redundancy of a neighbor cloud – enabling the recovery of protected virtual clusters on the corresponding neighbor cloud.

The Cloud Manager is provider specific (e.g. OpenStack heat, Kubernetes, Amazon EC2, etc.) and offers orchestration functionality to operate and provision virtual resources. Failures on the Cloud Manager are out of scope in this work.

Chapter 7

Conclusions and Future Work

"It always seems impossible until it's done."

Nelson Mandela

Fault Tolerance has become a crucial aspect to consider for the sustained growth of HPC. The previous chapters in this thesis have described the contributions made to offer a configurable multi-platform FT architecture for cluster and cloud environments. In this chapter we summarize the conclusions for each of the contributions and propose future research lines.

7.1 Fault Tolerance Manager with Distributed Checkpoints for Automatic Recovery

Fault tolerance solutions should offer high availability with flexibility and transparency when implementing high availability in user applications.

In this contribution, a Fault Tolerance Manager (FTM) is presented, to provide high availability to users in an automatic and transparent manner. FTM allows the implementation of FT in applications without significant user effort, but still enabling them flexibility in configurations such as: defining events in the applications when they want to take checkpoints, or time intervals. Furthermore, system administrators are not required to install any specific libraries to implement it.

The FTM extends RADIC architecture to support coordinated checkpoints and it adds a facility to replicate and distribute checkpoint files created on local storage to logical neighbors nodes.

The FTM architecture allows applications to perform automatic recover even when a computer node is lost. Multiple failures are supported, although simultaneous neighbor node failures are not. Furthermore, computation power could be limited in configurations without spare nodes, but the application can still end successfully. Also, this approach can minimize the human interaction and maintain an acceptable MTTR value, reducing execution costs when failures appear.

During the experimental evaluation, functionality of the FTM is validated in failure-free and failure scenarios. Results show that FTM with the distribution and replication facility, can protect the HPC applications execution using several storage configuration. The experiments also evidences a low impact on users application execution for private clusters and cloud environment.

7.2 Application-Layer FT with Dynamic Resource Controller

Fault Tolerance in the application-layer is a growing topic. A lot of improvements have been done to the existent solutions, regarding the storage, compression, and facility of the implementation.

This work leverages FTM into the application-level, combining existing FT techniques with a new designed and implemented dynamic resources controller. The solution uses a logger facility and application-level checkpoints that allows the uncoordinated and semi-coordinated recovery, avoiding to rollback and recovery all application processes in case of failures.

The dynamic controller tries to optimize the memory usage for FT protection task providing high availability to the application. In this work, experiments scenarios demonstrate the effectiveness of the presented controller in cases where the system ran out of memory. Furthermore, it also manage to let the application finish, without letting the application unprotected.

7.3 Configuring Fault Tolerance Protection and Recovery

The implementation of Fault Tolerance for the user's applications comes with a considerable overhead. This overhead depends on a series of factors regarding applications, system environments and user requirements.

This work contributes to the domain of Fault Tolerance by a design of an FPP model, which allows to identify where in the application execution is possible to start inserting FT protection, in order to get more benefits from FT. Furthermore, FPP can help users reduce overhead in certain scenarios and still maintain high availability. Although, it is not conceived to interfere in checkpoints created with non-FT purposes. FPP model has been shown to increase performance by up to 11% over just protecting the application with an interval model.

Regarding the FT recovery task, a model is proposed, to improve resource usage by designing a methodology for spare node configuration. It determines when a spare can be discarded for recovery purposes, reducing the resources cost function. This work improves the overall knowledge of the conditions, which influences the final execution time of the applications that experiences failures. This model can be applied either private clusters and cloud environments. The presented experiments allow concluding that in certain failure scenarios a spare node is not a mandatory resource, and as shown in the experimentation, to avoid the spare node is even more convenient than to setup it.

7.4 RaaS: Resilience as a Service for HPC in Cloud Environments

Cloud computing has disrupted the way computing is delivered to web-scale companies and telecommunications providers. HPC markets are also affected by such disruption as cloud evolves towards high-performance execution environments and specialized instances with usual features as *on-demand* scaling, agile deployment and provisioning of applications, and cost-efficient *pay-per-use* models. The challenge, however, is to provide HPC applications with fault tolerance capabilities that adapt and leverage these cloud features while remove the constrains of traditional HPC bare-metal systems.

In this work, we have introduced RaaS, a fault tolerance framework for HPC applications in cloud environments. This solution is offered as a service within the cloud ecosystem and benefits from the flexibility and diversity of virtual resources of clouds. Fault tolerance is achieved in a transparent, distributed and automated manner, requiring neither instrumentation from the user nor modifications to the application source code.

RaaS offers multiple alternatives to provide FT protection. The presented experiments show that users can optimize tradeoffs between costs and performance across different configuration options. Results were obtained using a real application as well

as HPC benchmarks in both failure-free and faulty systems. Results shows that the overhead reduction depends on the configuration alternatives in terms of number and type of checkpoint storage, which for the best scenario is up to 8%.

7.5 Future Work

This thesis tries to cover several aspects of fault tolerance configuration in bare-metal clusters and cloud environments. Although, it generates future work that can be tackled:

- Identify more factors that may have influence on the designed FT protection and recovery models. Extensions can be made, in order to consider specific fault tolerance task for semi-coordinated rollback-recovery.
- Further analysis can be done to identify the effects of the re-mapping processes when failures occur and the recovery is done.
- Analyze the possibility of extending RaaS architecture to support containerized platforms, and protect application executions against multiple failures in this kind of systems.
- Leverage RaaS FT architecture to support multiple availability zones with different cloud providers, in order to offer resilience to executions within hybrid clouds.
- Analyze and extend the multi-platform resilience manager (MRM), to support multiple message passing libraries, others than MPI, such as Charm++ for parallel applications execution.
- Perform weak and strong scalability analysis of the proposed MRM for parallel IO applications.

7.6 List of Publications

The research presented in this thesis has been published in the following papers:

1. **Jorge Villamayor**, Dolores Rexachs and Emilio Luque. *Configuring Fault Tolerance with Coordinated Checkpoint/Restart*, In "Jornadas de Paralelismo (SARTECO)", pp. 337-343, Córdoba, Spain, September-2015. [71]

This paper presents a study of the several configurations types for coordinated checkpoints, such as full and incremental, along with the invocation approaches, in order to adapt them to specific needs of the applications.

2. **Jorge Villamayor**, Dolores Rexachs and Emilio Luque. *Distributed Coordinated Checkpoints with Replication for Automatic Recovery*, In "Workshop on Parallel Programming for Resilience and Energy Efficiency at the PPOPP'16", Barcelona, Spain, July-2017. [71]

   This paper presents a technical strategy to provide users with an automatic recovery FT approach using coordinated checkpoints. The proposed strategy takes advantage of node's local storage in the execution runtime.

3. **Jorge Villamayor**, Dolores Rexachs and Emilio Luque. *A Fault Tolerance Manager with Distributed Coordinated Checkpoints for Automatic Recovery*, In "The 2017 International Conference on High Performance Computing and Simulation (HPCS)", pp. 452-459, Genova, Italy, July-2017. [72]

   This paper presents a Fault Tolerance Manager (FTM) for coordinated checkpoint files, which provides users automatic recovery from failures when losing computing nodes. This proposal makes the configuration of FT simpler and transparent for the users without requiring knowledge of their application implementation, nor the application's source code.

4. **Jorge Villamayor**, Dolores Rexachs and Emilio Luque. *When is the Right Time to Start the Fault Tolerance Protection?*, In "The 2017 International Conference on High Performance Computing and Simulation (HPCS)", pp. 426-433, Genova, Italy, July-2017. [73]

   This paper proposes a First Protection Point model, which determines the starting point to introduce FT protection gaining benefits in terms of total execution time including failures.

5. Diego Montezanti, **Jorge Villamayor**, Armando De Giusti, Marcelo Naiouf, Dolores Rexachs and Emilio Luque. *A Methodology for Soft Errors Detection and Automatic Recovery*, In "The 2017 International Conference on High Performance Computing and Simulation (HPCS)", pp. 434-441, Genova, Italy, July-2017. [51]

   This article proposes a methodology that improves system reliability against transient faults, when running parallel message-passing applications. The proposed

solution uses FTM [72], to cover silent errors by addressing the problem using multiple user-level checkpoints.

6. **Jorge Villamayor**, Diego Lugones, Dolores Rexachs and Emilio Luque. *RaaS: Resilience as a Service*, In "The IEEE/ACM International Symposium on Cluster, Cloud and Grid computing (CCGrid)", pp. 356-359, Washington, USA, May-2018. [75]

   This paper presents Resilience as a Service (RaaS), a fault tolerant framework for HPC applications running in cloud. In this paper RADIC architecture is used to provide clouds with a highly available, distributed and scalable fault-tolerant service.

7. **Jorge Villamayor**, Dolores Rexachs and Emilio Luque. *RADIC based Fault Tolerance System with Dynamic Resource Controller*, In "The International Conference on Computational Science (ICCS)", pp. 624-631, Wuxi, China, June-2018. [74]

   This paper introduces a Fault Tolerance Manager (FTM) implemented in the application-layer following the uncoordinated and semi-coordinated rollback recovery protocols. A dynamic resource controller is added to the FTM, which monitors the message logger buffers and performs actions to maintain an acceptable level of protection.

References

[1] Aarseth, S. J. (2003). *Gravitational N-body simulations: tools and algorithms*. Cambridge University Press.

[2] Ali, M. M., Southern, J., Strazdins, P., and Harding, B. (2014). Application level fault recovery: Using fault-tolerant open mpi in a pde solver. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1169–1178. IEEE.

[3] Amazon (2018). AWS, High Performance Computing. https://aws.amazon.com/hpc/. Accessed: 2018-09-11.

[4] Ansel, J., Arya, K., and Cooperman, G. (2009). DMTCP: Transparent checkpointing for cluster computations and the desktop. In *IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*, pages 1–12.

[5] Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., and Matsuoka, S. (2011). FTI: High performance Fault Tolerance Interface for hybrid systems. *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12.

[6] Bland, W. (2012). User level failure mitigation in mpi. In *European Conference on Parallel Processing*, pages 499–504. Springer.

[7] Bland, W., Bouteiller, A., Herault, T., Hursey, J., Bosilca, G., and Dongarra, J. J. (2013). An evaluation of User-Level Failure Mitigation support in MPI. *Computing*, 95(12):1171–1184.

[8] Bosilca, G., Bouteiller, A., Herault, T., Robert, Y., and Dongarra, J. (2015). Composing resilience techniques: Abft, periodic and incremental checkpointing. *International Journal of Networking and Computing*, 5(1):2–25.

[9] Bouteiller, A., Cappello, F., Dongarra, J., Guermouche, A., Hérault, T., and Robert, Y. (2013a). Multi-criteria checkpointing strategies: Response-time versus resource utilization. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8097 LNCS, pages 420–431.

[10] Bouteiller, A., Herault, T., Bosilca, G., and Dongarra, J. J. (2013b). Correlated set coordination in fault tolerant message logging protocols for many-core clusters. *Concurrency Computation Practice and Experience*, 25(4):572–585.

[11] Bouteiller, A., Herault, T., Krawezik, G., Lemarinier, P., and Cappello, F. (2006). MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI. *International Journal of High Performance Computing Applications*, 20(3):319–333.

[12] Buntinas, D., Coti, C., Herault, T., Lemarinier, P., Pilard, L., Rezmerita, A., Rodriguez, E., and Cappello, F. (2008). Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI Protocols. *Future Generation Computer Systems*, 24:73–84.

[13] Cabello, U., Rodriguez, J., Meneses, A., Mendoza, S., and Decouchant, D. (2014). Fault tolerance in heterogeneous multi-cluster systems through a task migration mechanism. *2014 11th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*, pages 1–7.

[14] Cao, J., Simonin, M., Cooperman, G., and Morin, C. (2015). Checkpointing as a Service in Heterogeneous Cloud Environments. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 61–70. IEEE.

[15] Cappello, F. (2009). Fault Tolerance in Petascale/Exascale Systems: Current Knowledge, Challenges and Research Opportunities. *COMPUTING APPLICATIONS The International Journal of High Performance Computing Applications*, 23(3):212–226.

[16] Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., and Snir, M. (2009). Toward exascale resilience. *The International Journal of High Performance Computing Applications*, 23(4):374–388.

[17] Castro-León, M., Meyer, H., Rexachs, D., and Luque, E. (2015). Fault tolerance at system level based on RADIC architecture. *Journal of Parallel and Distributed Computing*, 86:98–111.

[18] Chavarria-Miranda, D., Huang, Z., and Chen, Y. (2012). High-Performance Computing (HPC): Application & Use in the Power Grid. *Power and Energy Society General Meeting*, pages 1–7.

[19] Cheraghlou, M. N., Khadem-Zadeh, A., and Haghparast, M. (2016). A survey of fault tolerance architecture in cloud computing.

[20] Coti, C., Herault, T., Lemarinier, P., Pilard, L., Rezmerita, A., Rodriguezb, E., and Cappello, F. (2006). Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 18–18. IEEE.

[21] Cunningham, D., Grove, D., Herta, B., Iyengar, A., Kawachiya, K., Murata, H., Saraswat, V., Takeuchi, M., and Tardieu, O. (2014). Resilient X10. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '14*, volume 49, pages 67–80, New York, USA. ACM Press.

[22] Daly, J. T. (2006). A higher order estimate of the optimum checkpoint interval for restart dumps. In *Future Generation Computer Systems*, volume 22, pages 303–312.

[23] Dauwe, D., Pasricha, S., Maciejewski, A. A., and Siegel, H. J. (2017). An Analysis of Resilience Techniques for Exascale Computing Platforms. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 914–923. IEEE.

[24] Di, S., Robert, Y., Vivien, F., and Cappello, F. (2016). Toward an Optimal Online Checkpoint Solution under a Two-Level HPC Checkpoint Model. *IEEE Transactions on Parallel and Distributed Systems*, 9219(c):1–1.

[25] Diouri, M. E. M., Glück, O., Lefevre, L., and Cappello, F. (2012). Energy considerations in checkpointing and fault tolerance protocols. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 3–8.

[26] Duarte, A., Rexachs, D., and Luque, E. (2006). Increasing the cluster availability using RADIC. *2006 IEEE International Conference on Cluster Computing*, pages 1–8.

[27] Egwutuoha, I. P., Chen, S., Levy, D., and Selic, B. (2012a). A Fault Tolerance Framework for High Performance Computing in Cloud. *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2012)*, pages 709–710.

[28] Egwutuoha, I. P., Chen, S., Levy, D., Selic, B., and Calvo, R. (2012b). A Proactive Fault Tolerance Approach to High Performance Computing (HPC) in the Cloud. *2012 Second International Conference on Cloud and Green Computing*, pages 268–273.

[29] Egwutuoha, I. P., Levy, D., Selic, B., and Calvo, R. (2013a). Energy Efficient Fault Tolerance for High Performance Computing (HPC) in the Cloud. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 762–769. IEEE.

[30] Egwutuoha, I. P., Levy, D., Selic, B., and Chen, S. (2013b). A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326.

[31] Elliott, J., Hoemmen, M., and Mueller, F. (2014). Evaluating the impact of sdc on the gmres iterative solver. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1193–1202. IEEE.

[32] Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408.

[33] Expósito, R. R., Taboada, G. L., Ramos, S., González-Domínguez, J., Touriño, J., and Doallo, R. (2013). Analysis of I/O Performance on an Amazon EC2 Cluster Compute and High I/O Platform. *Journal of Grid Computing*, 11(4):613–631.

[34] Fialho, L., Rexachs, D., and Luque, E. (2011). What is missing in current checkpoint interval models? *Proceedings - International Conference on Distributed Computing Systems*, pages 322–332.

[35] Fialho, L., Santos, G., Duarte, A., Rexachs, D., and Luque, E. (2009). Challenges and issues of the integration of RADIC into open MPI. In Ropo, M., Westerholm, J., and Dongarra, J., editors, *Lecture Notes in Computer Science*, volume 5759 LNCS, pages 73–83. Springer.

[36] ForumMPI (2015). MPI: A Message-Passing Interface Standard. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. Accessed: 2018-09-11.

[37] Gainaru, A., Cappello, F., Snir, M., and Kramer, W. (2013). Failure prediction for HPC systems and applications: Current situation and open issues. *International Journal of High Performance Computing Applications*, 27(3):273–282.

[38] Gao, Q., Huang, W., Koop, M. J., and Panda, D. K. (2007). Group-based coordinated checkpointing for MPI: A case study on infiniband. *Proceedings of the International Conference on Parallel Processing*, (Icpp).

[39] Geist, A. (2016). How To Kill A Supercomputer: Dirty Power, Cosmic Rays, and Bad Solder - IEEE Spectrum. https://spectrum.ieee.org/computing/hardware/how-to-kill-a-supercomputer-dirty-power-cosmic-rays-and-bad-solder. Accessed: 2018-07-09.

[40] Geist, A. and Reed, D. A. (2017). A survey of high-performance computing scaling challenges. *The International Journal of High Performance Computing Applications*, 31(1):104–113.

[41] Gómez, A., Carril, L., Valin, R., Mouriño, J., and Cotelo, C. (2014). Fault-tolerant virtual cluster experiments on federated sites using BonFIRE. *Future Generation Computer Systems*, 34:17–25.

[42] Gupta, A., Faraboschi, P., Gioachin, F., Kale, L. V., Kaufmann, R., Lee, B.-S., March, V., Milojicic, D., and Suen, C. H. (2016). Evaluating and Improving the Performance and Scheduling of HPC Applications in Cloud. *IEEE Transactions on Cloud Computing*, 4(3):307–321.

[43] Hargrove, P. H. and Duell, J. C. (2006). Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing.

[44] Hursey, J. and Squyres, J. (2007). The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8.

[45] Hursey, J., Squyres, J., and Lumsdaine, A. (2006). A Checkpoint and Restart Service Specication for Open MPI. *Indiana University, Computer Science Department, Technical Report*.

[46] Jhawar, R., Piuri, V., and Santambrogio, M. (2012). A comprehensive conceptual system-level approach to fault tolerance in Cloud Computing. In *2012 IEEE International Systems Conference SysCon 2012*, pages 1–5. IEEE.

[47] Jhawar, R., Piuri, V., and Santambrogio, M. (2013). Fault tolerance management in cloud computing: A system-level perspective. *IEEE Systems Journal*, 7(2):288–297.

[48] Liu, X., Xu, X., Ren, X., Tang, Y., and Dai, Z. (2013). A Message Logging Protocol Based on User Level Failure Mitigation. In Kołodziej, J., Di Martino, B., Talia, D., and Xiong, K., editors, *International Conference on Algorithms and Architectures for Parallel Processing*, volume 8285 of *Lecture Notes in Computer Science*, pages 312–323, Cham. Springer International Publishing.

[49] Losada, N., Cores, I., Martín, M. J., and González, P. (2017). Resilient MPI applications using an application-level checkpointing framework and ULFM. *The Journal of Supercomputing*, 73(1):100–113.

[50] Meyer, H., Muresano, R., Castro-León, M., Rexachs, D., and Luque, E. (2017). Hybrid Message Pessimistic Logging. Improving current pessimistic message logging protocols. *Journal of Parallel and Distributed Computing*, 104:206–222.

[51] Montezanti, D., Giusti, A., Naiouf, M., Villamayor, J., Rexachs, D., and Luque, E. (2017). A methodology for soft errors detection and automatic recovery. In *Proceedings - 2017 International Conference on High Performance Computing and Simulation, HPCS 2017*, pages 434–441.

[52] Moody, A., Bronevetsky, G., Mohror, K., and Supinski, B. R. d. (2010). Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE.

[53] Mukwevho, M. A. and Celik, T. (2018). Toward a Smart Cloud: A Review of Fault-tolerance Methods in Cloud Systems. *IEEE Transactions on Services Computing*, pages 1–18.

[54] NASA (2018). NAS Parallel Benchmarks. https://www.nas.nasa.gov/publications/npb.html. Accessed: 2018-09-11.

[55] Oracle (2018). Bare Metal Cloud Services. https://cloud.oracle.com/bare-metal. Accessed: 2018-09-11.

[56] R Systems (2017). Dedicated Clusters. http://rsystemsinc.com/. Accessed: 2018-09-11.

[57] Rieker, M., Ansel, J., and Cooperman, G. (2006). Transparent user-level checkpointing for the native posix thread library for linux. In *PDPTA*, volume 6, pages 492–498.

[58] Ruscio, J. F., Heffner, M. A., and Varadarajan, S. (2007). Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE.

[59] Sabalcore (2018). High Performance Computing (HPC) in the Cloud. http://www.sabalcore.com/. Accessed: 2018-09-11.

[60] Sankaran, S., Squyres, J. M., Barrett, B., Sahay, V., Lumsdaine, A., Duell, J., Hargrove, P., and Roman, E. (2005). The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493.

[61] Santos, G., Duarte, A., Rexachs, D., and Luque, E. (2008). Providing non-stop service for message-passing based parallel applications with radic. In Luque, E., Margalef, T., and Benítez, D., editors, *Euro-Par 2008 – Parallel Processing*, pages 58–67, Berlin, Heidelberg. Springer Berlin Heidelberg.

[62] Sato, K., Moody, A., Mohror, K., Gamblin, T., Supinski, B. R. d., Maruyama, N., and Matsuoka, S. (2014). FMI: Fault Tolerant Messaging Interface for Fast and Transparent Recovery. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1225–1234. IEEE.

[63] Schroeder, B. and Gibson, G. A. (2007). Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1):012022.

[64] Schroeder, B. and Gibson, G. a. (2010). A Large-Scale Study of Failures in High-Performance Computing Systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350.

[65] Selikhov, A., Bosilca, G., Germain, C., Fedak, G., and Cappello, F. (2002). Mpich-cm: A communication library design for a p2p mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 323–331. Springer.

[66] Snir, M., Wisniewski, R. W., Abraham, J. a., Adve, S. V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., Chien, a. a., Coteus, P., DeBardeleben, N. a., Diniz, P. C., Engelmann, C., Erez, M., Fazzari, S., Geist, a., Gupta, R., Johnson, F., Krishnamoorthy, S., Leyffer, S., Liberty, D., Mitra, S., Munson, T., Schreiber, R., Stearley, J., and Hensbergen, E. V. (2014). Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28(2):129–173.

[67] StarCluster (2011). StarCluster documentation (v.0.95.6). http://star.mit.edu/cluster/index.html. Accessed: 2018-09-11.

[68] Suo, G., Lu, Y., Liao, X., Xie, M., and Cao, H. (2013). NR-MPI: A Non-stop and Fault Resilient MPI. *2013 International Conference on Parallel and Distributed Systems*, pages 190–199.

[69] Tan, L., Song, S. L., Wu, P., Chen, Z., Ge, R., and Kerbyson, D. J. (2015). Investigating the Interplay between Energy Efficiency and Resilience in High Performance Computing. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 786–796. IEEE.

[70] The Open Group (2015). The UNIX System – The Single UNIX Specification. http://www.unix.org/what_is_unix/single_unix_specification.html.

[71] Villamayor, J., Rexachs, D., and Luque, E. (2015). Configuring Fault Tolerance with Coordinated Checkpoint / Restart. *Jornadas de Paralelismo 2015*, pages 337–343.

[72] Villamayor, J., Rexachs, D., and Luque, E. (2017a). A fault tolerance manager with distributed coordinated checkpoints for automatic recovery. In *Proceedings - 2017 International Conference on High Performance Computing and Simulation, HPCS 2017*, pages 452–459.

[73] Villamayor, J., Rexachs, D., and Luque, E. (2017b). When is the right time to start the fault tolerance protection? In *Proceedings - 2017 International Conference on High Performance Computing and Simulation, HPCS 2017*, pages 426–433.

[74] Villamayor, J., Rexachs, D., and Luque, E. (2018a). Radic based fault tolerance system with dynamic resource controller. In Shi, Y., Fu, H., Tian, Y., Krzhizhanovskaya, V. V., Lees, M. H., Dongarra, J., and Sloot, P. M. A., editors, *Computational Science – ICCS 2018*, pages 624–631, Cham. Springer International Publishing.

[75] Villamayor, J., Rexachs, D., Luque, E., and Lugones, D. (2018b). RaaS: Resilience as a Service. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 356–359. IEEE.

[76] Vishwanath, K. V. and Nagappan, N. (2010). Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 193–204, New York, NY, USA. ACM.

[77] Wadhwa, A. and Bala, A. (2016). Preventing Faults: Fault Monitoring and Proactive Fault Tolerance in Cloud Computing. pages 665–673. Springer, Singapore.

[78] Wang, C., Vazhkudai, S., Ma, X., and Mueller, F. (2014). Transparent Fault Tolerance for Job Input Data in HPC Environments. http://optout.csc.ncsu.edu/~mueller/ftp/pub/mueller/papers/springer14.pdf.

[79] Wong, A., Rexachs, D., and Luque, E. (2015). Parallel Application Signature for Performance Analysis and Prediction. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):2009–2019.

[80] Zhang, Y., Zheng, Z., and Lyu, M. R. (2011). BFTCloud: A Byzantine Fault Tolerance Framework for Voluntary-Resource Cloud Computing. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 444–451. IEEE.

[81] Zhou, A., Wang, S., Cheng, B., Zheng, Z., Yang, F., Chang, R., Lyu, M., and Buyya, R. (2016). Cloud Service Reliability Enhancement via Virtual Machine Placement Optimization. *IEEE Transactions on Services Computing*, pages 1–1.

Appendix A

Execution Environments

A.0.1 Hardware Configuration

Three execution environment were setup for the experiments in this Thesis (Table A.1) with the configuration described as follows:

- A local cluster (AOCLSB-FT). Every node has a dual Gigabit ethernet interface. Each node can be accessed via 2 interfaces: one is exclusively to management tasks and other to perform computation. A NFS file server is accessed from nodes, and each node has a local disk with an EXT3 file system.
- A local cluster (AOCLSB-L), composed of nodes connected through Infiniband and Gigabit ethernet interfaces. The computation is performed using the Infiniband network, which has a peak of 40 Gbps, the management tasks are performed using the Gigabit ethernet interface. A NFS file server is accessed from nodes, and each node has a local disk with an EXT3 file system.
- A virtual cluster build with VMs acquired in a public well-known cloud computing Amazon EC2. The VMs are launched using StarCluster, an open source cluster-computing toolkit for Amazon EC2 designed by MIT. Each instance is a HVM (Hardware-assisted Virtual Machine), this is a virtualization type, which provides the ability to run an operating system directly on top of a virtual machine without any modification. All instances are rented from US East (N. Virginia) data center of Amazon.
- The bare-metal cluster used at Nokia Bell Labs to build the private cloud (GORO) environment using OpenStack is composed of 7 hosts. There is one host, which

acts as a controller and compute. The remaining 6 hosts are exclusively for compute in the installation.

Table A.1 Experimental Environments

COMP.	AOCLSB-FT	AOCLSB-L	AMAZON EC2	GORO
CPU	2 quad-core Intel(R) Xeon(R) E5430 @ 2.66 GHz	8 AMD Opteron (x8) Processor 6262 HE @ 1.60 GHz	8 (vCPU) Intel Xeon E5-2680v2, 2.8 GHz	Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz
RAM Memory	16 GB	252 GB	16 GB RAM	125 GB
Local Storage	30 GB HDD		2 x 80GB SSD	1080 GB HDD
Centralized Storage	NFS: 715 GB (v3)		NFS: 50 GB (v4) EBS (Elastic Block Storage)	—
Network	Dual Broadcom NetXtreme IITM 5708 Gigabit Ethernet; MI (Management Interface) and CI (Computation Interface)	CI: Infiniband (40Gbps); MI:Gigabit Ethernet	High Performance	Up to 1000 Mbps (full duplex)
Operative System	CentOS release 6.4	CentOS release 6.2	Ubuntu 11.10	CentOS release 6.2
Nodes	PowerEdge M600 (Dell)	PowerEdge C6145 (Dell)	c3.2xlarge	—