Recovery-Oriented Software Architecture for Grid Applications (ROSA-Grids)

A thesis submitted in fulfilment of the requirements for
the degree of Doctor of Philosophy

Iman Ibrahim Yusuf

B.CompSc.

School of Computer Science and Information Technology

College of Science, Engineering, and Health

RMIT University

March, 2012

To my husband—Seid Harun Osman
and my mother—Nebiha "Ebuye" Abdullahi
with love, gratitude and the utmost respect

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Iman Ibrahim Yusuf
School of Computer Science and Information Technology
RMIT University
30^{th} March, 2012

Acknowledgement

It is moments like this that all the words in a language do not seem sufficient enough to extend one's gratitude. Of course, when the language in question is your third one, then things could get a bit messy. I am extremely grateful to what Professor Heinz Schmidt, my first supervisor, has done for me. I thank him for believing in me from the moment I stepped into his office until now. He is one of the main reasons that I am finally able to write this chapter of the thesis. He taught me deep thinking, looking outside of the box, holding myself together when things do not go as planned, and many more things. Dr Ian Peake, my second supervisor, has also played a major role in shaping up my research and helping me stay focussed. His curiosity led to a lot of fruitful discussions, which many of them are reflected throughout the thesis. I cannot thank both my supervisors enough for helping me reach a major milestone in my life.

What can I say about Sheila Howell? Sheila figuratively hold my hand from the start to the end of my PhD project. I am grateful for everything Sheila has done for me, in particular, for her encouragement, unconditional help and reviewing my thesis. I would like to extend my sincere gratitude to Professor Lin Padgham for playing a significant role in making my life as a PhD student as smooth as I can hope for. I would like to also extend my appreciation to Dr Ian Thomas for reading my thesis and giving me helpful feedback. I would like to thank Dr Sebastian Sardina for showing me how to create beautiful latex posters.

During my PhD candidature, I met great people which added an extra dimension to my life. I would like to especially mention the PhD students with whom I started my post-graduate study: Dhirendra Singh, Carlos Alexandre Queiroz, Vidura Gamini Abhaya, Malith Jayasinghe, Nicholas May, Amir Aryani, and Steven Burrows. I also ended up meeting two great ladies: Palka "*Is this not Haram?*" Arora, and Naimah "*Would you like to join us for dinner?*" Yaakob. I consider myself lucky to have Palka and Naimah in my life. I would

like to also mention members of the Software Engineering discipline (formerly DSEA), and the staff members in the eResearch office. I especially would like to thank Sangeeta Devaraj for organising meetings with Professor Heinz, and providing useful administrative information. The relatively smooth execution of my experiments would not have been possible without the technical support of Anton "the Sumo Wrestler" Demidov, and Peter "the Comedian" Wolynec.

My family, in particular my mother and my husband, has played a tremendous role in removing every obstacle in front of me so that I can achieve what I have always desired. Considering what my mother, Nebiha "Ebuye" Abdullahi, has done for me, other than giving birth to me, I cannot literally construct any sentence to explain the extent of my love, gratitude and respect for her. My mother, an educated business woman who successfully raised three children by herself, is my role model. I can only hope to be as intelligent and strong as she is.

My husband, Seid Harun Osman, is the foundation of my life. He is my love, my counselor, my advisor, my caretaker, and my life-long partner. He is the man who sacrificed his career to make my dream come true. He is the man who put his ambitions aside to support my causes. He is the man who has been through thick and thin with me. I would like to extend my deepest gratitude to my husband for all the things that he has done for me. I love you, *R.B.*!

I would like to also thank my brother Omer Yusuf for being *cool* big brother. Speaking of cool relatives, I would like to mention my cousins Helen, Sarah, Yasmin, Elias and Zeki, with whom I shared quite memorable childhood. These guys and gals have been rooting for me until I touch the finishing line. I would like to also mention the next generation of our family; my sister Jemila, and my cousins Rinas, Mawerdy, and Sami **"ManUtd 8 - 2 Arsenal"** Fuad. I would like to thank my uncles and aunts, in particular, Fuad "Waf", Aba Remila, Amina, Dehab, Uma Asiya, Sayo, Ferial, Abaya "Mekia", Munir, Addus, Kemal, and their respective family members for keeping me in their prayers. I would like to also remember my friends, whom I consider as my sisters, Sarut Abdulkafur and Adanu.

The list of my relatives would not be complete without mentioning the Osman's family. I am especially grateful to Ahmed and his mother Zubeida for their constant encouragement. I would like to thank the rest of the family members: Osman "Shebaw", Fuad, Mohammed, Awad, Osman Osman, Fethia, Hayat and Ibrahim "Derbabaw" for welcoming me to their family with open arms.

Finally, the ultimate respect and gratitude is extended to the One Who, with His infinite kindness, has given me the strength to carry on regardless of the many curve balls life has thrown at me; to the One Who has surrounded me with wonderful people; and to the One Who let me see myself writing the last sentence of my thesis. Thank You, *Allah*!

Credits

Portions of the material in this thesis have previously appeared in the following publications:

- Iman I. Yusuf, Heinz W. Schmidt and Ian D. Peake. Architecture-Based Fault Tolerance Support for Grid Applications. *In the Proceedings of the Seventh International ACM SIGSOFT Conference on the Quality of Software Architectures (QoSA 2011)*, pages 177-181. Boulder, Colorado, USA, June 20–24, 2011.

- Iman I. Yusuf. Recovery-Oriented Software Architecture for Grid Applications. *In the Supplemental Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010).* Chicago, USA, June 28–July 1, 2010.

- Iman I. Yusuf, Heinz W. Schmidt and Ian D. Peake. Evaluating Recovery Aware Components for Grid Reliability. *In the Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009)*, pages 277-280. Amsterdam, the Netherlands, August 24–28, 2009.

This work was supported by the Endeavour International Postgraduate Research Scholarship.

Abstract

Grids are distributed systems that dynamically coordinate a large number of heterogeneous resources to execute large scale projects involving collaborating teams of scientists, high-performance computers, massive data stores, high bandwidth networking, and/or scientific instruments like telescopes, and synchrotrons. Failure in grids is arguably inevitable due to the massive scale and the heterogeneity of grid resources, the distribution of these resources over unreliable networks, the complexity of mechanisms that are needed to integrate such resources into a seamless utility, and the dynamic nature of the grid infrastructure that allows continuous changes to happen. To make matters worse, grid applications are generally long running, and these runs repeatedly require coordinated use of many resources at the same time.

Based on the traditional fault tolerance techniques that enable a system to complete its function even though the system and/or the environment where the system operates are faulty, the grid community has proposed various fault tolerance approaches for grid applications. However, these approaches are limited in at least one of the following ways: a) the fault tolerance support is directed to only grid applications whose execution units are independent of each other, b) only reactive strategies are used, c) the execution of a proactive strategy is not based on the current status of the computation and the current likelihood of failure in the execution environment but rather on the history of grid resources or the load of the execution environment, and d) fault tolerance strategies are applied at the application level.

In this thesis, we propose the *Recovery-Aware Components (RAC)* approach. The RAC approach enables a grid application to tolerate failure reactively and proactively at the level of the smallest and independent execution unit of the application. The approach also combines runtime prediction with a proactive fault tolerance strategy. By managing failure at the smallest execution unit, and combining runtime prediction with a proactive fault tolerance

strategy, the RAC approach aims at improving the reliability of the grid application with the least overhead possible. Moreover, to allow a grid fault tolerance manager fine-tuned control and trading off of reliability gained and overhead paid, this thesis offers an architecture-aware modelling and simulation of reliability and overhead. The thesis demonstrates for a few of a dozen or so classes of application architecture already identified in prior research, that the typical architectural structure of the class can be captured in a few parameters. The work shows that these parameters suffice to achieve significant insight into, and control of, such tradeoffs.

The contributions of our research project are a) the RAC approach, a prediction and an architecture based hybrid fault tolerance support for grid applications, b) the study of the usage of the RAC approach to improve the reliability of grid applications whose architecture can be classified as MapReduce or Combinational Logic, c) Markov models that represent the execution behaviour of MapReduce and Combinational Logic grid applications for reliability and overhead analysis, d) in-depth analysis of the impact of prediction inaccuracy on the reliability-overhead tradeoff of the RAC approach, and e) a parameterised experiment testbed for simulating the execution of a grid application with fault tolerance support that adapts the principles of the RAC approach.

We proposed the RAC approach, first and foremost, to improve the reliability of grid applications with the smallest overhead possible. Since the exact reliability-overhead tradeoff of the RAC approach depends on many factors, we evaluated the RAC approach by answering a specific set of questions. *What is the reliability-overhead tradeoff that is enabled by the RAC approach for MapReduce and Combinational Logic grid applications? How sensitive is such reliability-overhead tradeoff to a fault tolerance strategy and its parameters, and prediction accuracy?* Before embarking on answering these questions, we first define and formalize the concept of the RAC approach. The results of the reliability-overhead tradeoff evaluations are as follows. We have confirmed that, via simulated experiment, architecture based fault tolerance support provides better reliability improvement and incurs higher overhead to grid applications than the architecture unaware one. The degree of the superiority of the architecture aware fault tolerance support depends on factors like the type of the fault tolerance strategy selected and its parameters, and the accuracy of a predictor. Since runtime prediction is a central part of the RAC approach, we also evaluated the impact of prediction accuracy on the reliability-overhead tradeoff of the RAC approach. An increase in false positives, *predicting the presence of a non-existent failure*, increases reliability improvement; whereas an increase in false negatives, *not predicting the presence of failure*, decreases reliability improvement. The impact of prediction accuracy on overhead depends on the type of the fault tolerance support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"It is a tremendous act of violence to begin anything.
I am not able to begin.
I simply skip what should be the beginning."
- Rainer Maria Rilke

The term "grid" was coined by Ian Foster and others in the nineties in analogy to the electric power grid [Foster and Kesselman, 1999, pp. 17−21] to designate a distributed computing system for utility high-performance computing. Grids are concerned with "... coordinated resource sharing and problem solving in dynamic multi-institutional virtual organizations" [Foster et al., 2001]. In grids, massive resources are coordinated to execute large scale projects involving collaborating teams of scientists, high-performance computers, massive data stores, high-bandwidth networking, and/or scientific instruments like telescopes, synchrotrons and collidors. Grid resources are geographically distributed and thus belong to various administrative domains. Furthermore, these resources not only join and leave the grid network at any time but also change their access policy without any notice. As grid computing enters the mainstream and is applied in internet search, finance and banking, and large-scale engineering, the key issues of distributed systems, such as interoperability, security and fault tolerance, grow in importance. However, this research covers only *fault tolerance*.

The objective of this research is to improve the reliability of grid applications. Failure in grids is arguably inevitable due to (a) the massive scale and the heterogeneity of grid

resources, (b) the distribution of these resources over unreliable networks, (c) the complexity of mechanisms that are needed to integrate such resources into a seamless utility, and (d) the dynamic nature of the grid infrastructure which allows continuous changes to happen. To make matters worse, grid applications are generally long running and involve coordinating many resources simultaneously. Failure in such applications is therefore very costly as it requires restarting and rerunning previously completed computations, holding many resources for a long time repeatedly.

Various fault tolerance approaches have been proposed to increase the probability of successful execution of grid applications. Traditional fault tolerance techniques such as restart [Dean and Ghemawat, 2004], replication with/without voting [Chtepen et al., 2009, Budati et al., 2007], checkpointing [Nazir et al., 2009], migration [Kandaswamy et al., 2008], N-version [Xu et al., 2008], and a combination of these and other strategies [Hwang and Kesselman, 2003] have been tried. There are also grid tools that have built-in fault tolerance support. Examples of such tools include Condor-G, for managing a grid infrastructure [Frey et al., 2002], and Taverna, for building a grid workflow [Oinn et al., 2006]. Yet existing fault tolerance solutions are limited in such a way that they exhibit some of the following behaviours:

- Existing fault tolerance approaches either do not address the type of the grid application for which they are providing fault tolerance support (e.g., [Nazir et al., 2009]) or are explicitly proposed for a grid application whose execution units are embarrassingly parallel (e.g., [Chtepen et al., 2009]).

- Reactive strategies are primarily used for managing failure (e.g. [Dean and Ghemawat, 2004]). Reactive strategies attempt to recover a failed computation *after* the failure occurs while proactive strategies attempt to either minimize or prevent the impact of future failure on the overall computation *before* the failure occurs. Tolerating failure using only a reactive fault tolerance strategy could be very costly, especially if the failure occurs towards the end of a long running grid application execution.

- In cases where proactive fault tolerance strategies are used, existing fault tolerance approaches do not, in general, consider the *current* status of the computation and likelihood of failure in the execution environment before executing a proactive strategy. Instead, the history of grid resources and the load of the system are mainly used. For example, multiple copies of an execution unit are simultaneously executed if the execution envi-

ronment is *known* to be unreliable [Budati et al., 2007] or the load of the environment is *low* [Silva et al., 2003].

- Fault tolerance strategies are employed at the application level. Though grid applications are naturally composed of multiple and possibly long running execution units, this does not mean that significantly many of them will fail. Therefore, since enforcing a fault tolerance strategy at the application level will include the execution units that are not going to fail, application level fault tolerance introduces an unnecessary overhead. Duan et al. [2005], for example, showed the disadvantage of checkpointing the state of the entire execution, and the intermediate output of the execution that are needed to resume the computation in case of failure. The overhead of such type of checkpointing significantly increases with the size of the intermediate output.

In light of the limitations, inexpensive fault tolerance support for grid applications, which are composed of not only independent but also communicating execution units, is needed.

In this thesis, we propose the *Recovery-Aware Components (RAC)* approach. The RAC approach enables a grid application to tolerate failure reactively and proactively. In order to limit the extent of fault tolerance support *overhead*, which is the computational power that would be consumed by either the application to recover from failure and/or the fault tolerance management, the approach handles failure at the level of the smallest and independent execution unit of the grid application. The approach also combines runtime prediction with a proactive strategy to further reduce the overhead of the fault tolerance support. The purpose of the runtime prediction is to avoid an unnecessary execution of a proactive strategy. An execution unit will be replicated, for instance, only if either the unit or its execution environment is predicted to fail.

In order to cater to the fault tolerance requirements of grid applications whose communication and computation pattern is not classified as embarrassingly parallel, the RAC approach systematically uses the *class* of the architecture of the applications to provide customized fault tolerance support. For this, the classification of parallel programs by Asanovic et al. [2006, 2008, 2009] is used. Asanovic et al. [2006] classified parallel programs into thirteen computational kernels, known as *dwarfs*. Each dwarf has a different kind of parallel coordination, i.e. communication and computation pattern. For each coordination, one can assume a different capability of utilising the parallel structure to increase reliability and decrease cost of fault tolerance support. However, the actual reliability gain, cost reduction, and the constraints

under which these can be achieved, if at all, are far from obvious and require some methodical approach and evaluation.

We study the reliability-overhead tradeoff that is enabled by the RAC approach for a grid application whose parallel coordination can be classified under either the MapReduce or the Combinational Logic dwarf. The MapReduce dwarf represents grid applications that are executed in two distinct phases, identified as *map* and *reduce*. All execution units in the map phase are embarrassingly parallel, while the executions in the reduce phase involve some communication. Embarrassingly parallel grid applications are MapReduce applications without the reduce phase. Google's search is a notable example of a MapReduce application [Dean and Ghemawat, 2004]. The Combinational Logic dwarf, on the other hand, represents grid applications that involve dataflow networks of functions that operate on streams of very large amounts of data. These applications are common in cyclic redundancy checks, weather forecasting, logistics or content-based network routing, e.g. [NIST, 1999, 2001, Kuntschke et al., 2006, Wang and Rundensteiner, 2009].

The contributions of our research project are a) the RAC approach, a prediction and an architecture-based hybrid fault tolerance support for grid applications; b) the study of the usage of the RAC approach to improve the reliability of grid applications whose architecture can be classified as MapReduce or Combinational Logic; c) Markov models that represent the execution behaviour of MapReduce and Combinational Logic grid applications for reliability and overhead analysis; d) in-depth analysis of the impact of prediction inaccuracy on the reliability-overhead tradeoff of the RAC approach; and e) a parameterised experiment testbed for simulating the execution of a grid application with fault tolerance support that adapts the principles of the RAC approach.

The results of our research project are as follows. Via simulated experiment, we have confirmed that architecture-based fault tolerance support provides better reliability improvement, although with higher overhead, to both MapReduce and Combinational Logic grid applications than the architecture-unaware one. The degree of the superiority of the architecture-aware fault tolerance support depends on factors like the type of the fault tolerance strategy selected, e.g., checkpointing, and its parameters, e.g., cost of a checkpoint, and the accuracy of a predictor, i.e. false positives and false negatives. Since runtime prediction is a central part of the RAC approach, we also evaluated the impact of prediction accuracy on the reliability-overhead tradeoff of the RAC approach. An increase in false positives, *predicting the presence of a non-existent failure*, increases reliability improvement; whereas an increase in false negatives, *not predicting the presence of failure*, decreases reliability improvement. The impact of prediction accuracy

on overhead depends the type the fault tolerance support.

1.1 Research Questions

We propose the RAC approach, first and foremost, to improve the reliability of the grid applications with the smallest overhead possible. Therefore, we study the degree of the reliability improvement that would be achieved by adapting the RAC approach, and the overhead of such improvement by addressing the following research questions:

i. What is an adequate formal representation of the RAC approach for making reliability and overhead analyses?

ii. What is the reliability-overhead tradeoff that is enabled by the RAC approach for

   a) MapReduce grid applications?

   b) Combinational Logic grid applications?

iii. How sensitive is the reliability-overhead tradeoff of the RAC approach to

   a) the parameters of the fault tolerance strategy with which the RAC is paired, e.g. the impact of the probability of unrecoverable failure and the overhead of a single checkpoint in restart-based and checkpointing-based fault tolerance support, respectively?

   b) prediction accuracy, i.e. false positives and false negatives?

1.2 Overview of Our Research Methodology

Our research methodology includes three elements: *modelling*, *simulations*, and *real runs*. Figure 1.1 shows the interaction between these elements and the tools that are used in each:

i. *Modelling.* We use Discrete Time Markov Chains (Section 3.4.1) to model the behaviour of a grid application execution that tolerates failure according to the RAC approach. Our formal models are parameterised and based on the concept of Bulk Synchronous Parallel computing (Section 4.2). PRISM [Kwiatkowska et al., 2011] and Matlab [MathWorks Website] are used for model checking.

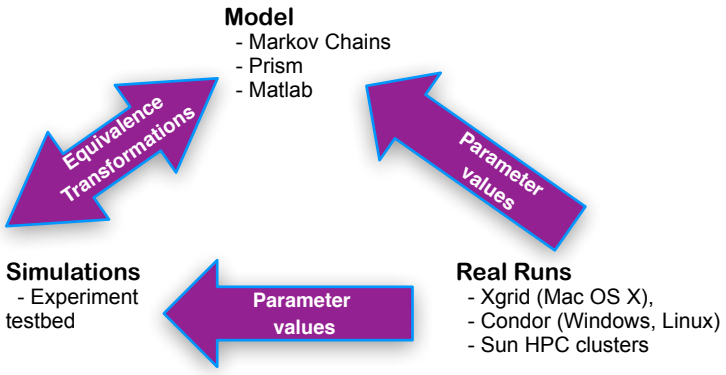


Figure 1.1: Dataflow between the three elements of our research methodology

ii. *Simulations.* The reliability-overhead tradeoff of the RAC approach is evaluated using experiments. For such evaluations, we designed and developed an experiment testbed. The testbed executes a grid application in either virtual or real time. During execution, the testbed provides a simulated fault tolerance support, which is based on the principles of the RAC approach, to the grid application. When the execution of the grid application is completed, the testbed outputs simulated reliability, simulated *execution time*, which is the real time that would have been elapsed from the start to end of the computation, and simulated overall *cost* of the execution, which is the total CPU time that would have been consumed by the computation. The experiment testbed is parameterised in order to allow the user to simulate the execution of a grid application with various fault tolerance strategies and configurations. In our evaluation, we obtained the values of some parameters, such as the time needed to complete an execution unit, from the output of real runs. Chapter 8 discuses the experiment design in detail.

iii. *Real runs.* We execute real applications to collect data for our evaluation. The data that we collect are the cost of communication, the time that is needed to complete an execution unit, the execution time of the overall computation, the cost of a fault tolerance strategy and others. These data, except for the overall execution time, are used as initial inputs during empirical evaluations. The overall execution time is used to evaluate the accuracy of our formal models and experiment testbed with respect to estimating the total time that is needed to complete the execution of a grid application. Our execution platforms are Xgrid, Condor, and Sun HPC clusters.

1.3 Thesis Scope

The scope of the thesis is as follows:

- We limit this study to only MapReduce and Combinational Logic grid applications ignoring the other eleven dwarfs for simplicity. This limitation does not restrict the generality of our approach, but would require further studies to confirm the exact nature of reliability-overhead tradeoffs for the other typical dwarfs underlying grid application not falling into these two classes of architecture.
- Estimating the effort that is required to implement a fault tolerant grid application based on the principles of the RAC approach is not part of the research project.
- In this thesis, as far as prediction is concerned, we focus on the impact of prediction accuracy on reliability and overhead, given a predictor. We explicitly exclude the study of prediction methods themselves. There exist extensive surveys about failure prediction, for example, by Salfner et al. [2010].

1.4 Thesis Outline

The rest of the thesis is organized into three parts:

- Part I
  - Chapter 2 introduces the reader to grid computing.
  - Chapter 3 discusses reliability and related topics.
  - Chapter 4 introduces software architecture, dwarfs, bulk synchronous parallel computing, the need for fault tolerance support in grids, existing fault tolerance approaches for grids and their limitations, and finally how these limitations could be addressed.
- Part II
  - Chapter 5 introduces the RAC approach and its architecture. In this chapter, we discuss how the different aspects of the RAC architecture are realised in a grid. We also present the abstract representation of the behaviour of the RAC architecture with respect to fault management. Such behaviour is formally modelled using Markov Chains.

  - Chapter 6 refines the RAC architecture to incorporate the requirements of a grid application based on the classification of the application's architecture. We study the customization of the RAC architecture for MapReduce or the Combinational Logic grid applications. The behaviour of the modified RAC architecture is formally modelled using Markov Chains.

- Part III

  - Chapter 7 presents our experiment testbed.
  - Chapter 8 describes our experiment design.
  - Chapter 9 presents the reliability-overhead tradeoffs that are enabled by the RAC architecture, and then shows how these tradeoffs are affected by the choice of a fault tolerance strategy, the parameters of the selected fault tolerance strategy, and the predictor's accuracy and interval.
  - Chapter 10 summarises our findings, and then indicates future research directions.

1.5 Summary

In this chapter, we motivated and introduced the research problem, that is, inexpensive fault tolerance support for a grid application that is composed of not only independent but also communicating execution units. We highlighted the limitations of existing fault tolerance approaches for grids, and then introduced the reader to our novel fault tolerance approach, which we refer to as the RAC approach. We also presented the research questions that we aim to answer at the end of this thesis. We provided readers a bird's-eye-view of our research methodology. We defined the scope of our research project. Finally, we briefly summarized the content of each chapter in the thesis. In the next part, Part I, we will introduce readers to the field of grid computing, reliability, and software architecture. Then, we will discuss our research problem and related issues in detail.

Part I

Background

"What you see is news, what you know is background, what you feel is opinion."
- Lester Markel

In this Part, we present the background theory that is related to our research project. The background theory is organised in three chapters. Grid computing is introduced in Chapter 2. Reliability and related topics are presented in Chapter 3. In Chapter 4, we discuss software architecture, dwarfs, bulk synchornous parallel model, the need for fault tolerance support in grids, existing fault tolerance approaches for grids, the limitations of these approaches, and finally how these limitations could be addressed.

Chapter 2

Grids

A *grid* [Foster et al., 2001, Foster, 2002] is a distributed system that dynamically coordinates a large number of heterogeneous resources, which are not under centralized control, using standard, open and general-purpose protocols and interfaces to provide desired qualities of service. Examples of grids are DAS-4 [DAS4 Website], DataTAG [DataTAG Website], and EU DataGrid [DataGrid Website]. *Grid computing* [Foster et al., 2001] is a large-scale distributed computing paradigm that is concerned with highly controlled and dynamically coordinated sharing of heterogeneous resources in multiple organizations.

Constituent resources of a grid, hereafter referred to as *grid resources*, come in many forms. Computers with various OS, programs, data, catalogues, code repositories, networks, sensors, HPC accelerators, and specialized equipments like synchrotrons [Australian Synchrotron, 2012] are all grid resources. Grid resources are owned by multiple individuals and/or organizations, and thus belong to various administrative domains. Grid resources are coordinated based on open and general-purpose protocols and interfaces that address the issues of authentication, authorization, resource discovery and resource access. Such protocols and interfaces enable the sharing of grid resources among collaborative individuals and/or organizations in a highly controlled manner.

Sharing in a grid is not limited to file exchange. Users have direct, but controlled, access to resources. A grid arranges the sharing of resources based on the specification of resource providers about "... what is shared, who is allowed to share, and the conditions under which sharing occurs" [Foster et al., 2001]. The sharing of resources in a controlled manner among different organizations and/or individuals, which aim to achieve a common goal, creates a *virtual organization*. Despite the resources in a virtual organization residing in multiple administrative domains, they can be discovered and accessed as though they all belong to a single

administrative domain.

Examples of a virtual organization include a common computational infrastructure for astronomers in multiple universities, heterogeneous and multi-organizational archival storage systems for large-scale multimedia content analysis, and a distributed platform to access weather models in different sites by a crisis management team to respond to an emergency situation. These and other virtual organizations differ from each other in terms of their purpose, scope, size and type of shared resources, structure, community and duration of their existence. Despite such disparity, virtual organizations share the following concerns and requirements [Foster et al., 2001]:

i. *Flexible sharing relationships*: In a virtual organization, sharing relationships change dynamically. Resources may leave an organization without any notification. The type of access to a particular set of resources or authorization methods may change. Therefore, there is a need to develop mechanisms for recognizing the current state of the virtual organization at any point of time.

ii. *Control over the usage of the shared resources*: Shared resources must be used according to the access policies of their providers. Therefore, there should be a mechanism to control what is shared, who is allowed to access the shared resources and in what way the shared resources are used.

iii. *Sharing of heterogeneous resources*: In a virtual organization, access to different types of resources is required. The heterogeneity in grid resources comes not only from their types but also from their configurations, architecture, and also access policies; for instance computers with Linux, Windows or Mac OS, and Windows laptops with AMD or intel processors.

iv. *Diverse usage mode*: Some grid resources participate in multiple sharing arrangements. Suppose a resource participates in two sharing arrangements: providing idle computing cycles to all members of a virtual organization, and allowing only members of a particular group a write access to its hard disk. If a resource is part of multiple sharing arrangements, it is not known how the resource will be used at a particular time. Therefore, there is a need for performance and other quality metrics to determine the exact usage of the resource.

A grid [Foster et al., 2001] uses protocols and services to address the above concerns and requirements of virtual organizations. Information protocols are used to learn the current

sharing relationship in a virtual organization. Security and management protocols control how, by whom and under which circumstances the shared resources are accessed. Job and data management protocols enable access to computing and data resources, respectively. These protocols ensure that the usage of a resource meets its quality metrics.

2.1 Grid Architecture

According to Foster et al. [2001], a *grid architecture* identifies system components that are needed to create, manage and exploit virtual organization sharing relationships among any potential members. A grid architecture also specifies the purpose of the components and their interactions. In this section, we study the Grid Protocol Architecture (Section 2.1.1) and the Open Grid Services Architecture (Section 2.1.2).

2.1.1 The Grid Protocol Architecture

The Grid Protocol Architecture is proposed by Foster et al. [2001]. The components of the architecture are protocols, which define the basic mechanisms to manage, discover, monitor, and access resources in a virtual organization. Figure 2.1* shows the Grid Protocol Architecture and its relationship with the Internet Protocol Architecture. The protocols of the Grid Protocol Architecture are divided into five layers: *Fabric*, *Connectivity*, *Resource*, *Collective* and *Application*.

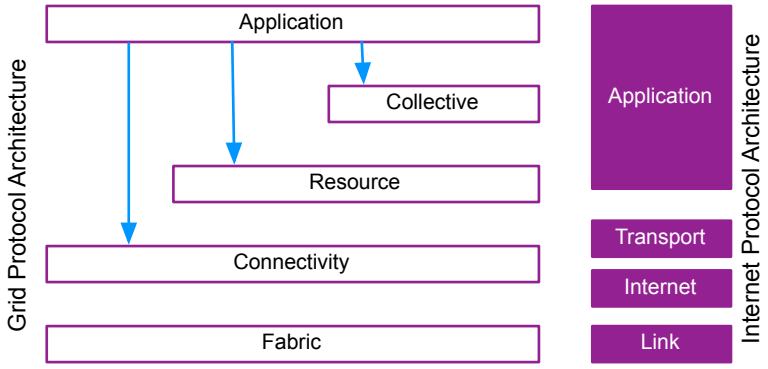


Figure 2.1: The layered Grid Protocol Architecture and its relationship with the Internet Protocol Architecture. This figure is adapted from Foster et al. [2001].

*Figure 2.1 is reprinted from *International Journal of High Performance Computing Applications*, 15, I. Foster, C. Kesselman, and S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations, 200-222, 2001, with permission from Sage Publications.

2.1.1.1 The Fabric Layer

The Fabric layer [Foster et al., 2001] provides access to grid resources that will be shared by members of a virtual organization. The sharing of these resources is mediated by grid protocols. Each resource is expected to at least implement enquiry and resource management mechanisms. Enquiry mechanisms are needed to obtain information about resources' structure and state, which include OS version, hardware configuration, available disk space, and network load. The resource management mechanisms control the quality of service that is provided by resources. Grid resources are mostly administrated by local resource managers, such as SGE, PBS [Bode et al., 2000] and Condor [Thain et al., 2005]. However, external managers are also used to add extra capability to resources. For example, GARA (Globus Architecture for Reservation and Allocation) [Foster et al., 1999] adds advance reservation capability to resources.

2.1.1.2 The Connectivity Layer

The Connectivity layer [Foster et al., 2001] provides communication and authentication protocols. The communication protocols are needed for exchanging data between grid resources. Therefore, these protocols support transport, routing, and naming. The communication protocols are assumed to come from the TCP/IP protocol stack.

The authentication protocols securely confirm the identity of users and resources. Authenticating users and resources in a virtual organization is characterized by *single sign on*: once users successfully sign in, they should have access to multiple resources without any further authentication; *delegation*: there should be a mechanism for users to let a program access resources on which the users have authorization; *integrations with local security solutions*: grid security solutions should find a way to seamlessly work with local security solutions; and *user-based trust relationships*: if users have access to resources from different administrative domains, the user should be able to access the resources without the need for interaction among the security administrators of the domains.

2.1.1.3 The Resource Layer

The Resource layer [Foster et al., 2001] is concerned with individual grid resources. This layer provides information and management protocols. Information protocols are for learning about a resource's configuration, load, usage policy, and other state and structure related information. Management protocols, on the other hand, are for negotiating access to a resource based on usage policy, instantiating sharing relationships, and monitoring and controlling the status of

the computation on the resource. Information and management protocols use Fabric layer functions to get information about a resource and manage the resource.

2.1.1.4 The Collective Layer

The Collective layer [Foster et al., 2001] is concerned with the global state of grid resources. Component of the Collective layer build on Resource and/or other Collective layers. The protocols and the services of this layer provide a range of sharing behaviours. Examples for sharing behaviours that are enabled by the components of the Collective layer abound. Members of a virtual organization can check the existence of resources via directory services. Members can also request a set of resources to be assigned for a particular purpose through co-allocation services. Monitoring and diagnostic services allow the user to oversee the health status of resources (alive vs. dead). Accounting services are used for billing. Collaboratory services facilitate information exchange among large groups of users.

2.1.1.5 The Application Layer

The Application layer [Foster et al., 2001] is the last and the top-most layer. This layer contains a program, hereafter referred to as a *grid application*, that users execute on the platform that is provided by virtual organizations. Examples of grid applications are image rendering, simulating the flow of blood through human arteries, large-scale multimedia content analysis, managing large datasets generated by particle accelerators, and visualizing earthquake simulation data. Since grid applications are executed in a distributed environment, their execution is naturally composed of multiple computations which potentially interact with each other. Hereafter, we refer to the smallest and independent computation unit of a grid application as an *activity* of a grid application.

2.1.2 The Open Grid Services Architecture

The *Open Grid Services Architecture (OGSA)* [Foster et al., 2002, 2005] is a service-oriented architecture that combines grid and Web Services technologies to define a set of service interfaces which are needed to create a standard-based grid. OGSA also defines semantics to specify the basic behaviours of a service, which include service creation, service naming, lifetime management, and communication. OGSA, however, does not impose the use of a specific programming language or execution environment during the implementation and execution

of services. We refer to a service whose interface and semantics are defined by OGSA as an *OGSA service*.

OGSA services integrate and manage heterogeneous resources within an organization and across multiple organizations. They also ensure that the desired quality of services are met during grid application execution in the presence of distributed and diverse resources. OGSA services are broadly categorised into six groups: *execution management*, *data*, *resource management*, *self-management*, *security*, and *information*.

Execution Management Services (OGSA-EMS) are concerned with managing computations. OGSA-EMS are in charge of finding and selecting resources that are suitable for the computation, setting up the execution environment, and initiating and overlooking the computation.

OGSA data services are concerned with managing data access and movement. These services use a virtualization interface, which obscures the difference between heterogeneous entities, to manipulate diverse data resources. OGSA data services transform data from one format to another, provide mechanisms to update data resources, and ensure a certain level of quality of service with respect to data delivery and integrity are met. OGSA data services also put an effort to minimize unnecessary data movement and duplication.

The Resource Management Services (OGSA-RM) look after grid resources in three ways. First, each resource is managed as an independent entity; for instance rebooting a computer. Second, resources are managed for being part of a grid. Under this management, OGSA-RM monitors and controls resources, and enables advance reservations whenever possible. The last management is concerned with the OGSA infrastructure; OGSA-RM monitors OGSA services like a registry service.

The Security Services are used for verifying the identity of users, identity mapping, controlling access to resources, ensuring privacy, and recording security-related events.

The purposes of Self-Management Services are to minimize the cost of owning grid resources and to simplify resource administration. These services are needed to make grid resources *self-configuring*: adapt to dynamic changes in their environment, *self-healing*: identify and resolve problems without disrupting their environment, and *self-optimizing*: adjust themselves to perform at a level where they can satisfy users' constraints.

The Information Services, the last category of OGSA services, are used for obtaining information about grid applications, grid resources and services. According to Foster et al. [2005], information refers to any logged data, and dynamic data and events that reveal the status of computations, resources and services.

Overall, OGSA services work together to provide a highly secured, controlled and seamless

environment for executing grid applications.

2.2 Grid Infrastructures

A *grid Infrastructure* is an environment where heterogeneous resources that reside on different administrative domains are dynamically shared in a highly controlled manner. Various tools are available in academic institutions, industries and other organizations for building a grid infrastructure. In this section, we highlight selected grid infrastructure tools, discuss the Globus Toolkit [Foster, 2006], which is arguably the most popular grid infrastructure tool, and finally define *the* Grid.

Tools like Alechmi, ALiCE, JCGrid, Condor-G, Nimrod/G, Oracle Grid Engine and Xgrid build a grid infrastructure where only *computing power* is shared. **Alchemi** [Luther et al., 2005] based grid infrastructures coordinate the sharing of computers with Windows OS. **ALiCE** [Teo and Wang, 2004] and **JCGrid** [Bucciarelli, 2012] are Java-based tools that build platform independent grid infrastructures. **Condor-G** [Frey et al., 2002] and **Nimrod/G** [Abramson et al., 2000, Buyya et al., 2000] are extended from their respective predecessors, Condor [Litzkow et al., 1997, Thain et al., 2005] and Nimrod [Abramson et al., 1995], by services that securely discover, access and manage resources in multiple administrative domains. Condor-G manages computation based on the mechanisms of its predecessor. Nimord/G [Abramson et al., 2000, Buyya et al., 2000] manages computation based on computational economy. In Nimrod/G, users are able to specify the deadline for completing the execution of their application and/or the price the users are willing to pay for the computation. **Open Grid Engine** [Ora, 2010] uses Service Domain Manager software to enable the sharing of computational resources among multiple administrative domains. In Open Grid Engine, computers with Windows OS, Mac OS X and Linux OS can be shared. **Xgrid** [App, 2009] provides an environment where computers with Mac OS X can be shared. Some works has been done to include non-Mac computers in Xgrid managed virtual organizations [Cote, 2004, Campbell, 2005].

DSpace, Gfarm, iRODS and OODT are tools for building a grid infrastructure for secure and controlled sharing, management and access of *data* across multiple collaborators. **DSpace** [The DSpace Developer Team, 2011] enables data sharing among collaborators over the web. **Gfarm** [Tatebe et al., 2010] based grid infrastructure coordinates local file systems on computers to enable data sharing. In order to improve I/O performance during the execution of data-intensive applications, Gfarm ensures that data access is either from local file system or from the nearby data storage. **iRODS** [iRODS Website] facilitates policy-based organization,

sharing, protection, and preservation of up to hundreds of millions of files. It also supports high-performance network data transfer. iRODS infrastructure can be built on Windows (without logical name-space), Linux, Unix, and Mac computers. **OODT** [OODT Website] based grid infrastructure enables collaborative data management and archiving. The infrastructure enables users to store data, and then search, retrieve and/or analyse the stored data. OODT is used in projects like the Early Detection Research Network [EDR, 2008], NASA's Planetary Data System [Jet, 2010] and the Orbiting Carbon Observatory [Crisp et al., 2003].

GLIDE and Globus are used to build grid infrastructures where multiple types of resources can be shared. **GLIDE** [Mattmann et al., 2005], which is the successor of OODT, allows building a grid infrastructure where both data and computational resources are shared. **Globus**, which provides *de facto* standard for building a grid infrastructure, creates an environment where computational, data and specialized resources are shared among multiple organizations.

2.2.1 The Globus Toolkit

The Globus Toolkit [Foster, 2006, Globus Website] is a collection of software libraries and services that provide support for developing grid applications, building grid infrastructures, and producing other support services. The Globus Toolkit is widely used in many successful projects like visualizing data at the Southern California Earthquake Center, simulating the flow of blood through human arteries at Brown University, and managing data at CERN and the Earth System Grid. The latest version of the Globus Toolkit, GT 5.2, was released on December 15^{th}, 2011.

Services in the Globus Toolkit follow the principles of OGSA. The core roles of these services include providing access to and managing computational and data resources, discovering resources that satisfy users' requirement, monitoring resources for detecting problems, controlling specialised equipments, transferring large data, authentication, authorization, and delegation. The libraries and the services of the Globus Toolkit are categorized as *job management components*, *data management components*, *information services components*, *security components*, and *common runtime components*.

Job management components are concerned with providing access to resources, and managing and monitoring executions on those resources. The Grid Resource Allocation and Management (GRAM) service, the Work Management Service (WMS), and the Grid Teleoperations Control Protocol (GTCP) service are job management components. GRAM provides interfaces for enabling access to, initiating executions on, and managing computational resources that

are located on remote sites. WMS provides the capability to dynamically generate execution sandboxes. GTCP provides interfaces for controlling specialised equipments like wave tanks.

Data management components are concerned with providing access to and transferring large datasets. Data management components of the Globus Toolkit are GridFTP, the Reliable File Transfer (RFT) service, the Replica Location Service (RLS), the Data Replication Service (DRS) and Data Access and Integration (OGSA-DAI) tools. GridFTP transfers large data between local and remote data storages at a high speed securely and reliably. RFT provides interfaces for reliable administration of more than one GridFTP transfer. RLS looks after and provides access to information about the storage sites of replicated data. DRS provides interfaces to manage data replication using GridFTP and RLS. OGSA-DAI provides a set of tools for accessing and processing relational and XML data.

Information services components, also known as the Monitoring and Discovery System (MDS), are concerned with providing information about the status and availability of grid resources. Index service, Trigger service, and WebMDS are information components. Index and Trigger services are aggregators that monitor resource and collect information. The two services differ the way they make information accessible. While Index service publishes the collected information at a specific location, Trigger service is event-driven and thus provides information only if the collected information satisfies a specific rule. WebMDS presents information via a web browser.

Security components, also known as the Globus Security Infrastructure (GSI), provide tools like MyProxy, GSI-OpenSSH and SimpleCA, and services like the Delegation service and the Community Authorization Service (CAS) for authentication, authorization, message protection, and delegation. These components provide message-level and transport-level security based on X.509 credentials. Message-level security is also provided using username-password combination. MyProxy enables users to securely get credentials whenever the need arises. GSI-OpenSSH facilitates the use of proxy credentials for single sign-on and file transfer. SimpleCA implements a certificate authority that issues X.509 certificates to users as well as services of the Globus Toolkit. The Delegation service is for delegating resources on the user's behalf and for credential renewal. CAS enables members of a virtual organization to make fine-grained policy about the usage of resources that are located in multiple sites.

The last set of components are common runtime components. These components are used for constructing containers that host Java, C, and Python web services.

Though the Globus Toolkit provides numerous tools for constructing a grid infrastructure, it is limited with respect to coordinating computations, data transfers and other activities.

However, higher-level coordination tools, which use the components of the Globus Toolkit for laying the foundation of the grid infrastructure, can be built. Nimrod/G and Condor-G are examples of such tools.

2.2.2 The Grid

Grid infrastructure development tools mostly use open and general-purpose but not *standard* protocols to build *a* grid. Due to the lack of standard "InterGrid" protocols, grids that are built using different technologies do not interoperate. Though Globus Toolkit protocols are considered *de facto* standard due to their widely usage, the issue of interoperability is not yet fully resolved. The grid community, in particular the Open Grid Forum [OGF], is actively working on the standardization of grid protocols. Once such standardization is completed, as Foster [2002] pointed out, any interested party that can speak standard grid protocols can join *the* Grid in the same way any computing machine that speaks internet protocols can be on the Internet. *The* Grid is, therefore, a distributed system that coordinates grid resources, which are under multiple administrative domains, using open, general-purpose and *standard* protocols to provide desired qualities of service.

2.3 Workflow in Grids

Some occasions necessitate the execution of multiple grid applications or services in a certain sequence, as requested by the user, to achieve a given goal. We refer to the execution of each grid application or service in such arrangement as a *task*. The output of a given task may be used as an input to another task. Some tasks are mutually exclusive, hence they run in parallel. The rest depend on one or more tasks, hence they run when the tasks on which they depend on are completed. The partially or fully automated execution of multiple tasks, where data is transferred between tasks when necessary, to reach the target goal is known as a *Workflow* [Hollinsworth, 1994]. By taking into consideration the current state of scientific workflows, which are described using sophisticated tools rather than complex shell scripts, and the current context of a grid, where services are increasingly being used to build its infrastructure, Fox and Gannon [2006] define a *Grid Workflow* as follows:

> "The automation of the processes, which involves the orchestration of a set of grid services, agents and actors that must be combined together to solve a problem or to define a new service".

Suppose a scientist wishes to render satellite images. In addition to executing the task that renders images, pre-rendering and post-rendering tasks need to be executed. The pre-rendering tasks include collecting images from satellite, filtering the images, and moving the filtered images to an accessible storage. Once the images are stored, they will be rendered. After rendering is completed, post-rendering tasks, such as data analysis and visualization, are executed. In this scenario, the grid workflow starts by collecting images from the satellite and ends by visualizing the rendered images. It is important to note that the scientist should explicitly specify which tasks are needed to achieve her goal, and in what order the tasks should be executed. The scientist may use either scripts or grid workflow management tools, like DAGMan [Couvares et al., 2007] and Taverna [Oinn et al., 2006], to express the dependency between her tasks.

2.3.1 Types of Grid Workflows

Grid workflows are classified into five groups based on their complexity [Fox and Gannon, 2006]:

i. **Linear workflows**: This is the least complex workflow. Tasks in this workflow are executed one after another. When a task is completed, its output will be transferred to the next task. If the execution time of all tasks in the workflow is small, simple scripts can be used to describe the workflow.

ii. **Acyclic graph workflows**: Acyclic graph workflows describe the execution order of tasks using graphs. The nodes of a graph represent tasks. The edges of a graph represent the execution order dependency between tasks. Some tasks are independent of other tasks, while others need to wait for one or more tasks to complete before they can be executed. Tools like DAGMan [Couvares et al., 2007] are used to describe an acyclic graph workflow. Since acyclic graph workflows represent the execution order of tasks, they are sometimes referred to as *composition in time*.

iii. **Cyclic graph workflows**: Cyclic graph workflows describe the dataflow dependency between services or component instances using graphs. It is possible for the services or component instances to stream data iteratively. Hence, the workflow is represented by cyclic graphs. The nodes of a cyclic graph represent a service, a component instance or an abstract model. The edges of a cyclic graph represent the message or the data to be passed between the nodes. Tools like Taverna [Oinn et al., 2006] are used to describe

a cyclic graph workflow. Due to the structuring of the workflow based on dataflow dependencies, a cyclic graph workflow is also referred to as *composition in space*.

iv. **A workflow of workflows**: If the structure of a workflow is too large and/or complex, describing the workflow using a graph may not be efficient. One way of representing a large and complex workflow is to decompose the workflow into smaller connected workflows, and then use a graph to show the dependency between the smaller workflows. The nodes and the edges of such graph represent a workflow and the (computation or data) dependency between the decomposed workflows, respectively. DAGMan and Kepler [Ludscher et al., 2006] allow the construction of a workflow of workflows. As pointed out by Fox and Gannon [2006], the complexity of workflows may arise due to change in the workflow during runtime, for instance removing a service (a node) in autonomic systems for optimization. Thus, mechanisms to intelligently deal with such situations are needed.

v. **Implicit graph workflows**: Implicit graph workflows are expressed in terms of desired outcomes. Any computational and/or data movement can be followed as long as the intended outcome is achieved.

2.3.2 Grid Workflow Management Systems

Grid Workflow Management Systems are software systems that coordinate the execution of multiple tasks on a grid infrastructure. These systems enable users to describe the dependency between the tasks using either plain text (e.g., DAGMan) or visual representation (e.g., Kepler). DAGMan [Couvares et al., 2007], Askalon [Fahringer et al., 2005], Pegasus [Deelman et al., 2005], Kepler [Ludscher et al., 2006], Taverna [Oinn et al., 2006], GridAnt [Amin et al., 2004], Triana [Taylor et al., 2007], and Gridbus workflow [Pandey et al., 2009] are examples of grid workflow management systems. In this section, we discuss DAGMan and Taverna.

DAGMan (Directed Acyclic Graph Manager) is a software system that manages the execution order of tasks, which correspond to either computation or data placement, on Condor infrastructure. While Condor and Stork are responsible for managing computational and data tasks, respectively, DAGMan ensures that each task is scheduled in an order as specified by the user. Users describe the dependency between tasks using scripts. DAGMan is suitable to manage plain and nested acyclic graph workflows. During the execution of a workflow, DAGMan records the progress of the computation. This enables users to monitor the execution

of their workflow from the log files. Similar to Condor and Condor-G, DAGMan is from the University of Wisconsin-Madison.

Taverna is a Java-based workflow management system, from the ^{my}Grid Project [myGrid Website], that provides a domain independent platform to design, build and execute scientific workflows. Taverna enables users to build a workflow which is composed of services with iteration, and thus it is suitable to manage cyclic workflows. Taverna provides command-line tools, and a rich and sophisticated graphical user interface for performing a wide range of activities, which include describing dependencies between services, validating and debugging dependencies, compositing multiple workflows, and pausing/resuming and tracking the progress of a workflow execution. Taverna is widely used in life science applications.

An extensive survey about more than a dozen grid workflow management systems is conducted by Yu and Buyya [2005]. The study compares and contrasts each system based on issues like workflow design, information retrieval, scheduling, fault tolerance and data movement.

2.4 Grid Computing vs. Cloud Computing

It has been and still is a hot topic, *grid computing vs. cloud computing*. People are still stumbling to understand the difference between the two worlds. Blogs, panel discussions (IEEE e-Science 2010 Conference), and peer-reviewed papers [Foster et al., 2008, Brandic and Dustdar, 2011] are dedicated to clarify what each world promises and delivers.

Both grid computing and cloud computing represent a large-scale distributed computing paradigm. They have a shared vision and common concerns. The ultimate goal behind these technologies is minimizing the cost of computing, data access, and data storage. Since both technologies enable multiple users to access a common pool of resources, they need to deal with privacy, confidentiality and resource management issues. Though grid computing and cloud computing are similar because of the paradigm they represent, the vision they share and the issues they are concerned with, as extensively discussed by Foster et al. [2008], each have their own business model, architecture, application model, security model, resource management, and programming model. Of the areas that separate the two technologies, we believe, the core difference between the two comes from their business models.

Grid computing is about *controlled* and *coordinated* dynamic resource *sharing* among *multiple* organizations, while cloud computing [Armbrust et al., 2010] is about *providing* highly *scalable* resources *on-demand*. In grid computing, the resources that are shared by users are owned by either the users or the organizations that the users represent; and each resource is

shared according to the usage policy of its owner. If new users (organizations) wish to access the resources in a given virtual organization, they are expected to bring new resources into the virtual organization. This way, the existing members of the virtual organization will have access to the resources of the new members, and vice versa. In cloud computing, however, an independent party, such as Amazon [EC2] and Google [Google App Engine], provides resources to users. All that is required to access resources is a credit card and an email address. Since users are charged based on their usage, they are free to either increase or decrease their usage at any time. In cloud computing, resources appear to be 'infinite'. Thus, no advance reservations are needed for provisioning.

The President and the CEO of the Open Grid Forum, Craig Lee, while responding to the untimely publication of the obituary of grid computing, summarised the difference between the two technologies as *"To sum it all up in one phrase - grids are about federation; clouds are about provisioning"*.

2.5 Summary

In this chapter, grid computing is introduced. We distinguished between a grid, grid computing and the Grid. We discussed the grid user concerns requirements that are addressed by grids. We presented two grid architectures, the Grid Protocol Architecture and the Open Grid Services Architecture (OGSA). Following this, we defined a grid infrastructure, gave some examples, and discussed the Globus Toolkit. We then introduced grid workflows, and briefly discussed workflow management systems. Finally, we presented the similarity and difference between grid computing and cloud computing.

Chapter 3

Reliability

Reliability is one of the attributes for measuring the degree to which a system can be trusted to carry out its intended function [Avizienis et al., 2004]. Reliability is a well-studied quality of a system. *Reliability*, according to the IEEE Standard Glossary of Software Engineering Terminology [IEEE, 1990], is

> "The ability of a system or component to perform its required functions under stated conditions for a specified period of time."

The quantitative definition of *Reliability* is as follows:

> "Software reliability is the probability of failure-free operation of a computer program for a specified time in a specified environment." [Musa et al., 1990, pp. 15],

The reliability of a software and a hardware system is affected by the environment under which each system operates. However, the principal cause that affects the reliability of each system is not the same. Design faults in a software system, and physical deterioration in a hardware system lead to reduced reliability. Despite the difference between the two systems, Musa [1998, pp. 35−36] argued that one can develop equivalent reliability theory for both systems. This is why the reliability of a software system can be measured using standard hardware combinatorial techniques.

3.1 Failure, Error, Fault

Failure, *error* and *fault* are dubbed as the "threats" to reliability of a system [Avizienis et al., 2004]. Despite the significance of these terms in reliability theory, they are sometimes sources

of confusion. Therefore, we distinguish these terms from each other using the standard IEEE definitions [IEEE, 1990], and the discussion of Avizienis et al. [2004].

A *failure* is the behaviour of a system that represents the deviation of the system from performing its required functions. Avizienis et al. [2004] defined failure as "... an event that occurs when the delivered service deviates from correct service". An *error* is the difference between the observed condition of a system when a failure occurs, and the correct condition of the system. Avizienis et al. [2004] defined error as the deviation of "... at least one (or more) external state of the system ... from the correct service state". A *fault* is a defect whose execution triggers an error. Avizienis et al. [2004] defined fault as the "... adjudged or hypothesized cause of an error". Failure is a user-oriented concept, and fault is a developer-oriented concept.

Here, we use an example to understand the difference between failure, error and fault. Suppose a robot should take five steps forward whenever a button is pressed. However, when the button is pressed, the robot takes only three steps. Since the robot does not move according to the specification, then pressing the button leads to a *failure*. The *error* in this scenario is the missing two steps. The *fault* is the line of code where the number of steps of the robot is incorrectly calculated.

3.2 Reliability Improvement Techniques

Numerous techniques are available to improve the reliability of a system. These techniques are broadly classified into four groups: *fault prevention*, *fault tolerance*, *fault removal*, and *fault forecasting* [Avizienis et al., 2004]. These techniques are also used to improve other system quality attributes like availability and safety.

Fault prevention techniques [Avizienis et al., 2004] are concerned with preventing the introduction of faults into a system. **Fault tolerance** techniques [IEEE, 1990] enable a system to perform its required function despite the presence of faults either in the system or in the environment where the system operates. **Fault removal** techniques [Avizienis et al., 2004] aim to remove as many exiting faults of a system as possible. Finally, **fault forecasting** techniques [Avizienis et al., 2004] predict existing and future faults of a system, and their consequences.

Each reliability improvement technique is broad and deserves to be explored further. However, since the main focus of the thesis is on fault tolerance, we investigate only fault tolerance techniques in detail.

3.3 Fault Tolerance

Fault tolerance is concerned with enabling a system to complete its function even though the system and/or the environment where the system operates are faulty. The goal of fault tolerance is, in other words, to avoid the failure of a system despite the presence of faults [Jalote, 1994, pp. 7]. According to the IEEE Standard Glossary of Software Engineering Terminology [IEEE, 1990], *fault tolerance* is

> "The ability of a system or component to continue normal operation despite the presence of hardware or software faults."

We refer to a system that performs its required function despite failure in some of its parts as a *fault tolerant system*. In general, any fault tolerant system manages faults in four phases: *error detection*, *damage confinement and assessment*, *error recovery*, and *fault treatment and continued system service* [Jalote, 1994, pp. 8−17].

i. **Error detection** [Jalote, 1994, pp. 8−14]: The first activity of a fault tolerant system is to detect an error. Error detection implies the presence of fault either in the system or in the environment where the system operates, and the failure of one or more components of the system. This is why the error detection phase is also known as the *fault/failure* detection phase. An error is detected by, for instance, comparing the outputs of equivalent system components, checking whether the time constraints are met, examining the internal structure of data, and checking whether the value of the output of a component is within an acceptable range. The effectiveness of a fault tolerant system highly depends on how good the error detection mechanism is. An ideal error detection mechanism should use the *specification* not the internal design of the system for detecting errors, detect *all* errors that the fault tolerant system intends to handle, never detect a *non-existent* error, and have a failure mode *independent* of the system. In practice, error detection mechanisms do not fully exhibit the characteristics of the ideal error detection mechanism.

ii. **Damage confinement and assessment** [Jalote, 1994, pp. 14]: Once the error is detected, the next phase is identifying the extent of its damage to the overall system computation. Since there is a delay between the time the system has failed and the time the error is detected, the error might propagate to other system components. In addition to the component from which the error is originated, the computation of other system

components could be jeopardized. Therefore, the extent of error propagation should be assessed either dynamically or statically. Dynamic assessment involves examining the information flow between components. Static assessment assumes the system to have barriers, beyond which no information flows. For instance, if an error is detected, then the extent of the damage is confined within two barriers.

iii. **Error recovery** [Jalote, 1994, pp. 15−16]: Once the error is detected and the extent of its damage is known, the next phase is to reach an error-free system state. This is achieved by removing the detected error by using either backward or forward recovery techniques. *Backward error recovery* restores the system to a known stable state. *Forward error recovery*, on the other hand, takes the computation forward by correcting the damage that is caused by the error to the overall system computation. If the cause of the error is a *transient fault*, a fault that exists for a bounded time, then it is expected that the fault will be gone by the time the error is successfully removed. On such occasions, no further fault tolerance management activities, such as fault treatment, are needed.

iv. **Fault treatment and continued system service** [Jalote, 1994, pp. 16−17]: The activities in this phase should be carried out if the cause of the error is a *permanent fault*, a fault that exists for unlimited duration. Even though the erroneous state of the system is corrected and the system is restarted from error-free state, if the cause of the error is a permanent fault, the error will re-occur unless the component which "hosts" the fault is bypassed. Therefore, in this phase, once the exact location of the fault is identified, the system will be repaired. System repair involves either not using the faulty component at all or substituting the faulty component by another component that could carry out the functions of the faulty component.

All activities in each phase should be done on-line. If there is manual intervention, then the system is not a fault tolerant system. The computation of a fault tolerant system could potentially be interrupted, especially when the system is being repaired in the fourth phase. Nonetheless, upon the completion of the fourth phase, or the third phase if the cause of the error is a transient fault, the system will continue its computation as if nothing had happened [Jalote, 1994, pp. 16].

3.3.1 Fault Tolerance Strategies

A *fault tolerance strategy* is a technique that a fault tolerant system uses either for error recovery, system repair or minimizing the impact of future failure on the overall system computation. We discuss widely-used fault tolerance strategies below.

- **Restart**: This is the simplest fault tolerance strategy. Restart resets a computation from the beginning when an error is detected. Restart can be applied locally or globally. Local restarts reset only the computation of system components that are affected by the error, while global restarts reset the entire computation of the system.

- **Checkpointing**: Checkpointing [Jalote, 1994, pp. 15] regularly saves the state of a system computation on a stable storage at predetermined intervals. This strategy is usually combined with other strategies like *roll-back* and *migration*.

- **Roll-back** Roll-back is used in conjunction with checkpointing. If the state of a system computation is checkpointed before an error is detected in the computation, then the computation will be rolled-back to the last stable checkpoint [Jalote, 1994, pp. 15].

- **Roll-forward**: Roll-forward takes the computation of a failed system forward by correcting the damage that is caused by an error(s) to the overall system computation [Jalote, 1994, pp. 15−16]. Roll-forward is also known as *forward error recovery*.

- **Migration**: Migration is combined with checkpointing and roll-back. When an event that may lead to error is detected in a checkpointed system, then the computation of the system will be migrated to a new execution environment [Litzkow et al., 1997]. In the new environment, the computation of the system will be restored from the last checkpoint.

- **Rejuvenation**: Rejuvenation is concerned with gradually terminating the computation of a system and then restarting or rolling-back the system immediately at potentially fault-free state [Huang et al., 1995]. The objective of rejuvenation is to minimize the impact of transient faults on the computation of a system.

- **Replication**: Replication is concerned with simultaneously executing multiple identical replicas of a system [Guerraoui and Schiper, 1997]. Replication serves two purposes: it almost guarantees at least one of the replicas will complete, and it enables voting based error detection mechanism by comparing the results of multiple replicas.

- **Redundancy**: Redundancy manages failure using primary-backup approach. Each system, usually a service or hardware, has a primary replica, and one or more backup replicas. During computation, the primary replica regularly sends its status to the backup replicas. In the event of the primary replica failure, one of the backup replicas takes the role of the primary replica. Redundancy is also known as *primary-backup replication* [Guerraoui and Schiper, 1997],

- **Standby spare**: Standby spare uses alternative or *standby* system components to ensure the computation of a system ends in success [Jalote, 1994, pp. 16]. If a system component fails, then it will be replaced by a component that can carry out the functions of the failed component.

- **N-version**: In the N-version fault tolerance strategy, the function of a system is implemented using N different methods [Avizienis, 1985]. All versions of the system are executed simultaneously. Then, the outputs of all or a subset of these executions will be examined to determine whether the system completes successfully or not.

3.3.2 Classification of Fault Tolerance Strategies

Fault tolerance strategies are broadly classified into *reactive* and *proactive* [Huang et al., 1995].

- Reactive fault tolerance strategies attempt to recover a failed system *after* the failure occurs. Restart and rollback are reactive fault tolerance strategies.

- Proactive fault tolerance strategies attempt to either minimize or prevent the impact of future failure on the overall system computation *before* the failure occurs. Checkpointing, replication, migration, and N-version are proactive fault tolerance strategies. These strategies are not executed to recover a failed system: checkpointing the state of a system minimizes the loss of computation time if the system fails later; replication and N-version increase the chance of successful system computation by executing multiple instances of the system, and migration takes the system to a new environment where the system could potentially complete its execution in success.

Redundancy and standby spare have both reactive and proactive elements. In redundancy, the backup replicas are executed to tolerate possible *future* failures. This action is a proactive technique. However, taking the role of the primary replica by one of the backup replicas is a reactive technique. This is because such activity occurs *after* the failure of the primary replica.

Similarly, in standby spare, incorporating alternative components into the fault tolerant system is a proactive action; however executing one of the alternatives in place of a failed component is a reactive action.

3.4 Reliability Prediction

Predicting the reliability of a software system has been studied for quite a while [Moranda, 1975, Cheung, 1980, Schmidt, 2003, Reussner et al., 2003, Brosch et al., 2011]. A *software system* is a collection of logically independent entities, known as *components*, that perform certain tasks [IEEE, 1990, Szyperski, 2002]. Components of a system are composed and coordinated solely via their interfaces or service contracts, which are accessible at run-time and distinguish required and provided services. Components have replaceable and independently deployable realizations.

Reliability prediction is concerned with estimating the reliability of a system under the context of certain usage profile and/or execution environment. The usage profile of a system describes the input parameters to the components of the system, how frequently each component is executed, and the interaction between the components [Reussner et al., 2003, Brosch et al., 2011]. If a component that is prone to failure is used frequently, for instance, then the likelihood of system failure increases. In addition to the usage profile, the reliability of a system is affected by its execution environment. Any external party, hardware or software, which the system interacts with could potentially cause failure in the system.

For predicting the reliability of a system, the operational behaviour of the system needs to be represented in a certain way. It is common to model the behaviour of a system using Markov chains [Cheung, 1980, Reussner et al., 2003, Wang et al., 2006, Cheung et al., 2008] or UML-like notations [Cortellessa et al., 2002, Brosch et al., 2011]. We refer to a system model that is primarily designed to predict reliability as a *reliability model*. If the reliability model of a system is a Markov chain (Section 3.4.1), then formal reliability analysis is performed using equations which describe the various behaviours of Markov chains. If the reliability model is a UML diagram [Booch et al., 2005], then the model is first transformed into an equivalent formal model, like Markov models, and then the reliability analysis is done using the formal model. The transformation of a UML-based reliability model to an equivalent formal model is in large part hidden from the user. Tools, such as PRISM [Kwiatkowska et al., 2011], Matlab [MathWorks Website] and RADL [Schmidt, 2003, 2007, Peake and Schmidt, 2011], provide an environment for analysing system reliability.

Due to the popularity of Markov chains for modelling system reliability and the need to ultimately transform UML-based models to equivalent formal models, we chose Markov chains for modelling the behaviour of a fault tolerant grid system in Chapters 5 and 6. Therefore, we introduce Markov chains to the reader in Section 3.4.1, and then discuss how Markov chains are used for reliability analysis in Section 3.4.2.

3.4.1 Markov Chains

A *Markov chain* is a stochastic process that has a discrete state space, and possesses the Markov property [Stewart, 2009, pp. 193−195].

Definition 3.1 (Stochastic process). A collection of random variables $\{X_t, t \in T\}$ that are defined on some probability space is called a *stochastic process* [Stewart, 2009, pp. 194]. The set T is called the index set or the parameter space of the process, and $T \subseteq \mathbb{R}$. The parameter t represents time. X_t is the *state* of the process at time t.

Definition 3.2 (Discrete state space). In a stochastic process $\{X_n, n \in \mathbb{R}\}$, the set of all possible states creates the *state space* of the stochastic process. If all states in the state space are discrete, then the process is called a *chain* and the state space is called a *discrete state space* [Stewart, 2009, pp. 194]. The elements of a discrete state space are identified by natural numbers. Let S denote a discrete space, $S \subseteq \mathbb{N}_0$.

Definition 3.3 (The Markov property). The *Markov property* [Stewart, 2009, pp. 193] states that, given the current state of a process, the future state of the process is conditionally independent of the previous states of the process. The future state of the process depends only on the current state of the process.

Definition 3.4 (Discrete-Time Markov Chains). A Markov chain in which a transition between two states occurs or fails to occur at discrete time steps, which are considered to be one unit apart, is called a *Discrete-Time Markov Chain (DTMC)* [Stewart, 2009, pp. 195]. The parameter space of a DTMC, denoted by T, is discrete, and thus $T = \{0, 1, 2, \dots\}$.

A DTMC is, therefore, a stochastic process $\{X_n, n \in \mathbb{N}_0\}$ that has a discrete state space $S = \{s_0, s_1, s_2 \dots\}$, and satisfies the Markov property. Let $X_n = s_{i_n}$ denote a DTMC in state s_i at time n, thus the Markov property of the DTMC is expressed as follows:

$$P\{X_{n+1} = s_{i_{n+1}} | X_n = s_{i_n}, X_{n-1} = s_{i_{n-1}}, \dots, X_0 = s_{i_0}\} = P\{X_{n+1} = s_{i_{n+1}} | X_n = s_{i_n}\} \quad (3.1)$$

Definition 3.5 (Time-Homogeneous Markov Chains). A Markov chain is said to be *time-homogeneous* if the transition probability between two states is independent of the time the transition is started. The transition probability from state s_i to state s_j in a time-homogeneous Markov chain is expressed as follows:

$$P\{X_{n+1} = s_{j_{n+1}} | X_n = s_{i_n}\} = P\{X_{n+m+1} = s_{j_{n+m+1}} | X_{n+m} = s_{i_{n+m}}\}, \quad \text{for } n = 0, 1, 2, \ldots \text{ and } m \geq 0 \tag{3.2}$$

Notation:

- We assume time-homogeneous DTMCs for modelling the behaviour of a fault tolerant system in Chapters 5 and 6. Therefore, unless otherwise specified, any reference to a Markov chain is a reference to a time-homogeneous DTMC throughout the thesis.
- Since our models contain a finite number of states, we limit our discussion to Markov chains with finite state space.
- Hereafter, a state of a Markov chain is identified by a single letter like i instead of by s_i.

Definition 3.6 (Transition Matrix). The transition probabilities between the states of a Markov chain are arranged in a matrix. The ij^{th} element of such matrix represents the transition probability from state i to state j. This matrix is called the *transition matrix* [Stewart, 2009, pp. 195].

Let P be the transition matrix of a Markov chain. The ij^{th} element of P is given by Equation (3.3).

$$P(i, j) = P\{X_{n+1} = j | X_n = i\} \tag{3.3}$$

P has the following two properties:

i. The entries in the transition matrix are probabilities. Therefore, $0 \leq P(i, j) \leq 1$.

ii. Each row in the transition matrix represent the transition probability distribution of a state in the Markov chain. Therefore, for all i, $\sum_{all\ j} P(i, j) = 1$.

Equation (3.4) shows the transition matrix of a Markov chain with m states, $m \geq 1$.

Definition 3.7 (Transition Diagram). A *transition diagram* [Stewart, 2009, pp. 197] depicts a Markov chain in a graphic form. The states of the Markov chain are represented by circles with labels, and the transitions between the states are represented by arrows, which are decorated with transition probabilities. Figure 3.1 shows equivalent representation of a Markov chain by a transition matrix and a transition diagram.

$$P = \begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \vdots \\ \vdots \\ \mathbf{m-1} \end{array} \begin{array}{c} \begin{array}{cccccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{\ldots} & \mathbf{\ldots} & \mathbf{m-1} \end{array} \\ \begin{pmatrix} \text{P(0,0)} & \text{P(0,1)} & \text{P(0,2)} & \ldots & \ldots & \text{P(0,m-1)} \\ \text{P(1,0)} & \text{P(1,1)} & \text{P(1,2)} & \ldots & \ldots & \text{P(1,m-1)} \\ \text{P(2,0)} & \text{P(2,1)} & \text{P(2,2)} & \ldots & \ldots & \text{P(2,m-1)} \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ \text{P(m-1,0)} & \text{P(m-1,1)} & \text{P(m-1,2)} & \ldots & \ldots & \text{P(m-1,m-1)} \end{pmatrix} \end{array} \tag{3.4}$$

$$P = \begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} \end{array} \\ \begin{pmatrix} 0 & 0.45 & 0.35 & 0.2 \\ 0.3 & 0 & 0 & 0.7 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

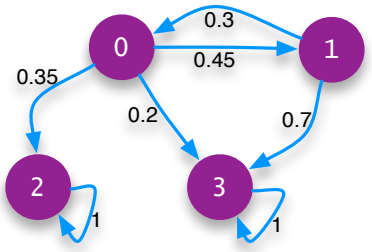


Figure 3.1: Equivalent Markov chain representation by a transition matrix and a transition diagram

Definition 3.8 (k-step transition probability). The transition probability of entering state j from state i after k intermediate transitions is called the *k-step transition probability* [Stewart, 2009, pp. 204]. Let P be the transition matrix of a Markov chain. $P(i,j)^{(k)}$ is the k-step transition probability from state i to j, and is shown in Equation (3.5).

$$P(i,j)^{(k)} = P\{X_{n+k} = j | X_n = i\}, \quad \text{for } k > 0 \tag{3.5}$$

$P(i,j)^{(1)}$, or just $P(i,j)$, is called the *single-step transition probability*.

Example 3.1 (Weather Prediction). Suppose the daily weather pattern of Melbourne is modelled using a Markov chain. The weather in Melbourne has three types, i.e. *windy*, *chilly* and *sunny*, and we assume each weather type is observed for full day. Therefore, the state space is $S = \{windy, chilly, sunny\}$. The transition matrix, denoted by P, with single-state transition probabilities is shown in Equation (3.6).

$$P = \begin{array}{c} \\ \mathbf{windy} \\ \mathbf{chilly} \\ \mathbf{sunny} \end{array} \begin{array}{c} \begin{array}{ccc} \mathbf{windy} & \mathbf{chilly} & \mathbf{sunny} \end{array} \\ \begin{pmatrix} 0.25 & 0.4 & 0.35 \\ 0.3 & 0.3 & 0.4 \\ 0.2 & 0.05 & 0.75 \end{pmatrix} \end{array} \tag{3.6}$$

From P, we observe that a windy day is followed by another windy day with 0.25 probability, or a chilly day with 0.4 probability, or a sunny day with 0.35 probability. However, what if we would like to know the probability of the weather to be sunny after two days, given today is windy? In order to answer questions like this, the *Chapman-Kolmogorov Equations* are used.

Definition 3.9 (Chapman-Kolmogorov Equations). The *Chapman-Kolmogorov equations* [Stewart, 2009, pp. 202−206] provide a method, shown in Equation (3.7), to compute the $k-$step transition probabilities.

$$\begin{aligned} P(i,j)^{(k)} &= \sum_{all\ r} P\{X_l = r | X_0 = i\} \times P\{X_k = j | X_l = r\} \\ &= \sum_{all\ r} P(i,r)^{(l)} \times P(r,j)^{(k-l)}, \quad \text{for } 0 < l < k \end{aligned} \tag{3.7}$$

Let $P^{(k)}$ be the transition matrix of a Markov chain after k transitions. The matrix notation of the Chapman-Kolmogorov equations is given in Equation (3.8).

$$P^{(k)} = P^k, \quad \text{for } 0 < k \tag{3.8}$$

Recall our question in Example 3.1, *what is the probability of the weather to be sunny after two days, given today is windy?* To answer this question, we set $k = 2$, and compute $P^{(2)}$. The result is shown in Equation (3.9). The answer is 0.51.

$$P^{(2)} = \begin{array}{c} \\ \textbf{windy} \\ \textbf{chilly} \\ \textbf{sunny} \end{array} \begin{array}{c} \begin{array}{ccc} \textbf{windy} & \textbf{chilly} & \textbf{sunny} \end{array} \\ \begin{pmatrix} 0.25 & 0.24 & 0.51 \\ 0.245 & 0.23 & 0.525 \\ 0.22 & 0.13 & 0.65 \end{pmatrix} \end{array} \tag{3.9}$$

Definition 3.10 (Transient and Recurrent States). Let $F(j,j)$ be the probability of ever returning to state j after leaving it. If $F(j,j) < 1$, then state j is called a *transient* state. If $F(j,j) = 1$, then state j is called a *recurrent* state [Stewart, 2009, pp. 208−209].

There is a non-zero probability for the Markov chain to never to return to a transient state. Therefore, transient states are visited finite number of times, Equation (3.10). In Figure 3.1, states 0 and 1 are transient.

$$\sum_{k=0}^{\infty} P(j,j)^{(k)} < \infty \tag{3.10}$$

The Markov chain certainly returns to a recurrent state. Therefore, recurrent states are visited infinitely often, Equation (3.11). In Figure 3.1, states 2 and 3 are recurrent.

$$\sum_{k=0}^{\infty} P(j,j)^{(k)} = \infty \tag{3.11}$$

Definition 3.11 (The Potential Matrix). Consider matrix R. The ij^{th} element of R is the expected number of times that the Markov chain visits state j, provided that state i is the first state of the Markov chain. R is called the *potential matrix* [Stewart, 2009, pp. 218−221].

Suppose T denotes the transition matrix of a Markov chain, and the ij^{th} element of T denotes the transition probability from transient state i to transient state j, for all transient states i and j in the Markov chain. The expected number of visits to the transient states of the Markov chain is computed as shown in Equation (3.12).

$$R = (I - T)^{-1} \tag{3.12}$$

T and R of the Markov chain in Figure 3.1 are given in Equations (3.13) and (3.14).

$$T = \begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \end{array} \begin{array}{c} \begin{array}{cc} \mathbf{0} & \mathbf{1} \end{array} \\ \begin{pmatrix} 0 & 0.45 \\ 0.3 & 0 \end{pmatrix} \end{array} \tag{3.13}$$

$$R = \begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \end{array} \begin{array}{c} \begin{array}{cc} \mathbf{0} & \mathbf{1} \end{array} \\ \begin{pmatrix} 1.16 & 0.52 \\ 0.35 & 1.16 \end{pmatrix} \end{array} \tag{3.14}$$

Consider $R(i,j)$. If state j is a recurrent state, the expected number of visits to state j from state i is shown in Equation (3.15).

$$R(i,j) = \begin{cases} \infty, & \text{if } P(i,j)^{(k)} > 0 \text{ and } k > 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.15}$$

Definition 3.12 (Absorbing States). An *absorbing state* [Stewart, 2009, pp. 207] is a recurrent state in which the Markov chain remains forever after the initial visit. Given an absorbing state i, $P(i,i) = 1$. In Figure 3.1, states 2 and 3 are absorbing states.

Definition 3.13 (The Absorption Probability Matrix). Consider matrix A. The ij^{th} element of A is the probability of ever reaching absorbing state j from transient state i. A is called the *absorption probability matrix* [Stewart, 2009, pp. 223−226].

Suppose B denotes the transition matrix of a Markov chain, and the ij^{th} element of B denotes the transition probability from transient state i to absorbing state j, for all transient states i and all absorbing states j in the Markov chain. The probability of ever reaching an

absorbing state from transient states of the Markov chain is computed as shown on Equation (3.16).

$$A = R \times B, \qquad \text{for } R \text{ in Equation (3.12)} \tag{3.16}$$

B and A of the Markov chain in Figure 3.1 are given in Equations (3.17) and (3.18).

$$B = \begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \end{array} \begin{array}{c} \begin{array}{cc} \mathbf{2} & \mathbf{3} \end{array} \\ \begin{pmatrix} 0.35 & 0.2 \\ 0 & 0.7 \end{pmatrix} \end{array} \tag{3.17}$$

$$A = \begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \end{array} \begin{array}{c} \begin{array}{cc} \mathbf{2} & \mathbf{3} \end{array} \\ \begin{pmatrix} 0.4 & 0.6 \\ 0.12 & 0.88 \end{pmatrix} \end{array} \tag{3.18}$$

3.4.2 Reliability Prediction with Markov Chains

Markov chains are one of the methods for modelling the behaviour of a system. Here, we use a classical reliability model, which was proposed by Cheung [1980], to discuss how the behaviour of a system with respect to reliability can be constructed and analysed using Markov chains.

The properties of the Cheung reliability model are given below:

- The reliability model represents the components of a given system by transient states. For a system with n components, there are n transient states. The execution of the system commences in the *entry* state, and ends in the *exit* state.
- The reliability model has two absorbing states C and F. State C represents successful execution of the system, while state F represents failure in the execution. If any of the components fails, the entire system computation is considered as failed.
- The arrow between states represents the direction of the flow of the computation.
- For all states i and j, let P_{ij} be the probability of executing the component in state j right after the component in state i completes execution, and R_i be the reliability of the component in state i. The probability of transiting from state i to state j is $P_{ij} \times R_i$.
- For all states i, since the failure of one component is suffice for the entire system to fail, there is a transition from state i to state F with the probability of $1 - R_i$.
- Given an exit state j, the system enters state C from state j with the probability of R_j.
- The reliability of the system is the probability of transiting from the entry state i to state C in the absorption probability matrix of the reliability model.

Example 3.2 (The Cheung Reliability Model). Suppose a system has four components, and the behaviour of this system with respect to reliability is modelled using a Markov chain. The reliability model of the system, denoted by M, has the properties of the Cheung reliability model. Figure 3.2 and Equation (3.19) show the transition diagram and the transition matrix of M, respectively.

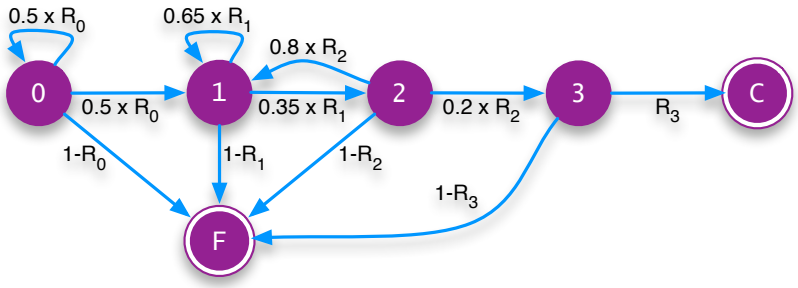


Figure 3.2: Example: Reliability model

$$M = \begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{F} \\ \mathbf{C} \end{array} \begin{array}{c} \begin{array}{cccccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{F} & \mathbf{C} \end{array} \\ \begin{pmatrix} 0.5 \times R_0 & 0.5 \times R_0 & 0 & 0 & 1 - R_0 & 0 \\ 0 & 0.65 \times R_1 & 0.35 \times R_1 & 0 & 1 - R_1 & 0 \\ 0 & 0.8 \times R_2 & 0 & 0.2 \times R_2 & 1 - R_2 & 0 \\ 0 & 0 & 0 & 0 & 1 - R_3 & R_3 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{3.19}$$

- The transient states of M are states 0, 1, 2 and 3. States 0 and 3 are the entry and the exit states, respectively.
- The absorbing states of M are states C and F. The selfloop transition of the absorbing states is removed, and these states are represented with double circles.
- The reliability of state i is denoted by R_i. For all states i and j, the probability of executing the component in state j right after the execution of the component in state i is given with concrete numbers. For example, the probability of transiting from state 1 to state 2 is $0.35 \times R_1$.
- All of the transient states of M transit to state F.

- The successful execution of the system is represented by the transition from state 3 to state C.
- Let $R_0 = 0.95$, $R_1 = 0.98$, $R_2 = 0.96$ and $R_3 = 0.99$. The reliability of the system, which is the transition from state 0 to state C in the absorption probability matrix of M, is 0.59. The absorption probability matrix of M is shown in Equation (3.20).

$$A = \begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \end{array} \begin{array}{c} \begin{array}{cc} \mathbf{F} & \mathbf{C} \end{array} \\ \begin{pmatrix} 0.41 & 0.59 \\ 0.35 & 0.65 \\ 0.31 & 0.69 \\ 0.01 & 0.99 \end{pmatrix} \end{array} \tag{3.20}$$

3.5 Recovery-Oriented Computing

The last reliability topic we introduce the reader to is Recovery-Oriented Computing (ROC). Despite the topic not being directly related to reliability, ROC focuses on system availability, the principle behind ROC is valuable and could be used to improve system reliability with relatively small cost.

Availability is the probability that a system is operational and accessible when it is needed [IEEE, 1990, Jalote, 1994, pp. 37−38]. Equation (3.21) shows the availability of a system, which is computed from the mean time to failure (MTTF) and the mean time to repair (MTTR) of the system.

$$\text{Availability} = \frac{MTTF}{MTTF + MTTR} \tag{3.21}$$

Recovery-Oriented Computing is concerned with increasing the availability of a system by decreasing the mean time to repair [Patterson et al., 2002, Fox and Patterson, 2005]. As evidenced by Equation (3.21), the availability of a system can be improved by either increasing the mean time to failure or decreasing the mean time to repair. The reason for the ROC to focus on MTTR rather than on MTTF is the belief that "*... hardware faults, software bugs, and operator errors are facts to be coped with, not problems to be solved*" [Patterson et al., 2002]. No matter how big MTTF is, system failure is inevitable.

Fast recovery after failure, in addition to increasing system availability, decreases revenue lost due to downtime. Furthermore, fast recovery reduces total cost of ownership of systems.

Patterson et al. [2002] argued that, since system administrators spend significant amount of time repairing systems, being able to repair failed systems quickly decreases the cost of administration.

3.6 Summary

In this chapter, reliability and related topics are presented. We defined reliability; distinguished between failure, error and fault; and then presented the four techniques for improving the reliability of a system. The four techniques are fault prevention, fault removal, fault tolerance and fault forecasting. We then dived deep into fault tolerance. We discussed the four phases that a fault tolerant system goes through to manage failure. We presented the common fault tolerance strategies and their classifications, i.e., reactive and proactive. We also showed fault tolerance strategies, such as redundancy, that have both reactive and proactive aspects. Following this, we discussed how to predict system reliability, introduced Markov chains, and presented the Cheung reliability model. Finally, we presented Recovery-Oriented Computing.

Chapter 4

Reliable Grid Software Design

Software architecture was recognised as being one of the fundamental disciplines of software engineering in the 1990s [Garlan and Shaw, 1994]. This discipline is concerned with high-level organization of the components of a software system. Such high-level system description allows study of the behaviour of a system with respect to, for instance, performance, reliability, and availability [Gokhale et al., 1998, Reussner et al., 2003, Brosch et al., 2011].

Garlan and Shaw [1994] and the IEEE Recommended Practice for Architectural Description of Software-Intensive Systems [IEEE, 2000] define *Software Architecture* as follows:

> "a collection of computational components—or simply *components*—together with a description of the interactions between these components—the *connectors*" Garlan and Shaw [1994]

> "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution." [IEEE, 2000]

Software architecture is in particular needed in systems in which a large number of components are integrated to complete certain tasks. This is because the main design challenges of a complex system come from neither algorithms nor data structures, but rather from the structuring of the components of the system. Such challenges include, but are not limited to, identifying components and their interactions, selecting communication and data access protocols, integrating components in a scalable manner, avoiding performance bottlenecks, and selecting the right design pattern from available architectural choices [Shaw and Garlan, 1996, pp. 1].

Architectural Style	Computational Model	Component	Connector	Invariant	Advantages	Disadvantages
1. Pipes and Filters [Garlan and Shaw, 1994]	A component reads streams of data, processes the data incrementally, and then outputs the (partial) result. **Example:** Unix programs, parsing in traditional compilers	Filters	Pipes	- Filters are independent units, and are oblivious about the identity of other filters; - The correctness of execution output is independent of the order of filters' incremental processing	- Understanding the overall system behaviour is possible due to the simplicity of filter composition; - Adding new filters or substituting old filters with new ones is easy, - Support reuse and concurrent execution.	- Not good for interactive programs - Performance loss if filters maintain communication between related streams or data transmission is based on the lowest common denominator protocol.
2. Data Abstraction and Object-Oriented Organization [Garlan and Shaw, 1994]	A component encapsulates data and procedures, provides interfaces though which other components can invoke its procedures. **Example:** Enterprise JavaBeans (EJB), CORBA	Objects	Function and procedure invocation	- An object manages its internal structure; - The internal structure of an object is not visible to other objects.	- Changing the internal structure of an object will not affect other objects.	- An object must know the identity of other objects in order to interact with them - Change in the identity of an object necessitates the modification of all other objects that interact with the object.
3. Event-based, Implicit Invocation [Garlan and Shaw, 1994]	The invocation of a procedure is associated with a specific event. When a component announces an event, all procedures that are associated with that event will be invoked. **Example:** Model-View-Controller	Modules with a set of events and procedures	Procedure calls, event and procedure call bindings	- Event announcers are oblivious about which and how other modules will be affected by the announced event	- Reuse. Any module can be part of the system by registering to a specific event. - Replacing a module by another one does not affect other modules in the system.	- An event announcer does not have any control over the order of procedure invocation, and does not know when a procedure is completed. - Sharing a repository among modules has performance and resource handling implications
4. Layered Systems [Garlan and Shaw, 1994]	Components are placed on top of each other. A component provides a service to the adjacent component above, and is served by the adjacent component below. **Example:** Grid protocol architecture, OSI protocol suite	Layers	Protocols	- Layers interact only with adjacent layers (top and bottom).	- Enable to decompose complex problems, and rearrange in order of increasing complexity - Modifying a layer affects only its adjacent layers. - Changing the internal implementation of a layer does not affect other layers	- Hard to structure systems in layered style - Hard to find the right abstraction. Higher level functions, when implemented, may span multiple layers.

Table 4.1: Examples of Architectural Styles

Some systems have similar types of components, and component interactions—and therefore share common architectural design. A family of systems that share a set of components, connectors, and constraints, which define the interactions among components, is called an *architectural style* [Garlan and Shaw, 1994]. In addition to components, connectors and constraints, an architectural style is identified by its underlying computational model, invariants, advantages, disadvantages, and common examples and specializations [Garlan and Shaw, 1994]. Some software systems have *heterogeneous* architecture, which is composed of more than one architectural style. Pipes and filters, data abstraction and object-oriented organization, event based implicit invocation, and layered systems are examples of architectural styles. Table 4.1 summarizes the properties of these architectural styles.

4.1 Dwarfs

A *dwarf*, also known as a *motif* [Asanovic et al., 2008], is a high level algorithmic abstraction that captures the pattern of communication and computation of parallel applications [Asanovic et al., 2006]. Parallel applications that belong to a specific dwarf have similar structure of computation and data movement, but possibly different implementations and computations. Inspired by the work of Phil Colella [Colella, 2004], who identified seven numerical methods for scientific computing, researchers at the UC Berkeley categorized existing parallel applications into thirteen dwarfs. The researchers proposed the use of these dwarfs to evaluate future parallel programming models and hardware architectures instead of traditional benchmarks such as SPEC (Standard Performance Evaluation Corporation) [SPE] or SPLASH (Stanford Parallel Applications for Shared Memory) [Woo et al., 1995]. Each dwarf is briefly explained in Table 4.2, which is based on discussions by Asanovic et al. [2006] and in Dwarf Mine Website. Note that the word "Grid" in the context of this section refers to a set of lines that cross with each other to form rectangles.

Some parallel applications are composed of multiple dwarfs [Asanovic et al., 2006]. Route lookup, for instance, is composed of the Graph Traversal and the Combinational Logic dwarfs. For an application with multiple dwarfs, the computation is distributed either *temporally* or *spatially*. In *temporal* distribution, computation is done as a sequence of dwarfs. The dwarfs are executed one after another, and all available resources are allocated to the currently running dwarf. In *spatial* distribution, the dwarfs communicate while running concurrently. On such scenarios, available resources are divided among the dwarfs. A dwarf could be executed using both types of distributions, for instance pipeline parallelism in the Combination Logic

dwarf (Section 6.2.1.1.2).

Table 4.2: The thirteen dwarfs of parallel applications. This table is based on discussions by Asanovic et al. [2006] and in Dwarf Mine Website.

Dwarf	Description	Example
1. Dense Linear Algebra	**-** Data are dense matrices or vectors in 3 levels: vector-vector, matrix-vector and matrix-matrix. Row and column data are read using unit-stride memory accesses and strided accesses, respectively.	Video compression
2. Sparse Linear Algebra	**-** Matrix-based applications with many zeros; data can be compressed by removing the zero entries for efficient use of storage and bandwidth.	Spring models
3. Spectral Methods	**-** Data are in frequency domain; use multiply butterfly stages; some stages are local while others are global with all-to-all communication.	Spectral clustering
4. N-Body Methods	**-** Computations depend on interactions between discreet points in a grid. In particle-particle methods, every point depends on every other point on the grid; and in hierarchical methods, forces are combined from many, but not from all, points.	Molecular dynamics
5. Structured Grids	**-** Data are arranged in a regular grid. Points are updated together using values from their immediate neighbours. Updates could be in-place, 2 version or alternate like red-black pattern. Has spacial locality; a subgrid can be executed on a separate processing element.	Finite element methods
6. Unstructured Grids	**-** Data are arranged in a mesh, which is composed of points, edges, faces and/or volumes. Each mesh element is updated together. Updating a mesh element necessitates identifying its neighbours and loading the neighbour's values. Due to the diversity of mesh elements, each element should be represented unambiguously.	Belief propagation
7. MapReduce	**-** MapReduce applications have two sets of computations: *map*, processes input data and produces an intermediate data, and *reduce*, processes the intermediate data and produces the final output. Computations are *embarrassingly parallel*. **NB**: Asanovic et al. [2009] reclassified MapReduce as a structural pattern, rather than a computational pattern.	Google's search
8. Combinational Logic	**-** Carry out simple functions, such as boolean operations, on a large set of data. Resembles Multiple-Instruction-Single-Data of Flynn's Taxonomy [Flynn, 1972]	Route lookup
9. Graph Traversal	**-** Main functionality is to walk through elements of a dataset and inspect the features of the elements. Minimal computations.	Decision trees

continued . . .

. . . continued

Dwarf	Description	Example
10. Dynamic Programming	- Build the optimal solution to a problem from the optimal solutions of simpler overlapping subproblems.	Query optimization
11. Backtrack and Branch-and-Bound	**-** The goal is to find a globally optimal solution by searching intractably large spaces. Divide and conquer is used to partition the search space, and explore each region independently.	Chess
12. Graphical Models	**-** Construct graphs, where the nodes and the edges represent random variables and conditional dependencies, respectively.	Bayesian Networks
13. Finite State Machine	**-** Consists of a set of states. Computation progresses by transiting from one state to another. Computations are *embarrassingly sequential*.	Text processing

4.2 Bulk Synchronous Parallel Model

Inspired by the von Neumann model of sequential computation, Valiant [1990] proposed the *Bulk Synchronous Parallel (BSP)* model, which serves as a bridge between software and hardware, for parallel computation. As the von Newmann model was able to unify the diverse software and hardware world of sequential computing, the purpose of the BSP model is to achieve such unity in the world of parallel computing. This way, hardware manufacturers can focus on developing BSP computers without being concerned about the type of programs that run on their machines. Likewise, software developers write parallel programs without explicitly considering the type of hardware on which their program runs, other than the hardware being a BSP computer.

Valiant [2011] extended the basic BSP model, which does not impose any memory restrictions, to include multiple memory and cache levels. Nonetheless, further discussions are based on the basic BSP model.

A BSP computer has three attributes: *components* that engage in processing and memory functions; a *router* that transports messages between components; and *synchronising facilities* that coordinate all or some of the components at a regular interval. In a BSP computer, a computation is modelled as a sequence of *supersteps*. Each superstep, as shown in Figure 4.1, is composed of local computations, global communications and a barrier synchronisation [Valiant, 1990, Skillicorn et al., 1997].

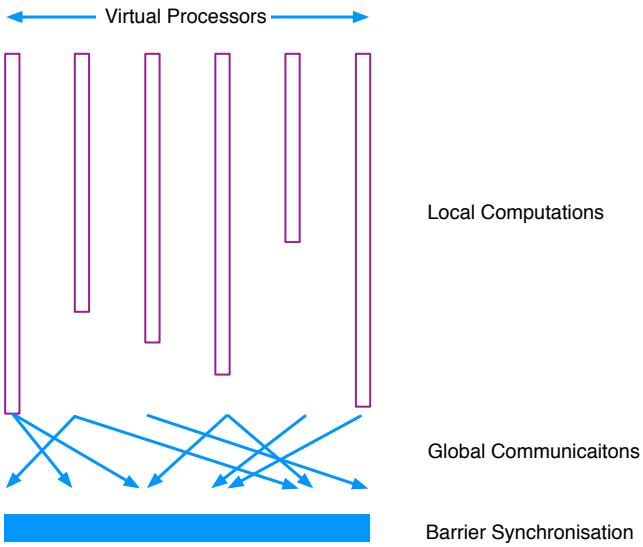


Figure 4.1: A superstep. This figure is adapted from Skillicorn et al. [1997].

The local computations in a superstep are independent of each other, and therefore can be executed in any order. During execution, the BSP components use only the data that is locally available to them. Therefore, before the start of a superstep, the required data for the local computations should be stored in a place to which the BSP components have access.

Once all of the local computations in a superstep are completed, then the BSP components communicate and exchange information, as needed. Each BSP component has incoming and outgoing messages. In a superstep, if the maximum number of messages that are either incoming to or outgoing from a BSP component is h, then the communication pattern of the superstep is known as an *h-relation*. The superstep in Figure 4.1* has a 2-relation communication pattern.

The last element of a superstep is a barrier synchronisation. The barrier synchroniser, upon the completion of all global communications, makes the data that is needed for the next superstep available in the local memory of the BSP components.

4.2.1 The BSP Cost Model

The execution time of a BSP program, as shown on Equation (4.1), is the sum of the execution time of its supersteps. The execution time of a superstep, as shown on Equation (4.2), is

*Figure 4.1 is reprinted from *Scientific Programming*, 6, D. Skillicorn, J. Hill, and W. McColl, Questions And Answers About BSP, 249-274, 1997, with permission from IOS Press.

constructed from the execution time of the longest local computation, the cost of the global communication of an h-relation, and the cost of the barrier synchronisation [Skillicorn et al., 1997].

Let

- S: the total execution time of a BSP program
- s_i: the execution time of superstep i
- $w_{i,j}$: the execution time of a local computation j in superstep i
- h_i: the h-relation of superstep i
- g: the transmission capacity of the network to deliver data
- b: fixed (amortised) cost of synchronisation

$$S = \sum_{i=1}^{k} s_i \qquad \text{for } k \text{ total supersteps in } S \tag{4.1}$$

$$s_i = \max_{j=1}^{n} w_{i,j} + h_i \times g + b \qquad \text{for } n \text{ local computations in superstep } i \tag{4.2}$$

Equation (4.2) shows the standard BSP cost model. However, other cost models are also available. For example, in Equation (4.3), local computation is merged with the cost of communication. However, Skillicorn et al. [1997] argued that using such finer cost models would not change the final result by more than a small constant factor.

$$s_i = \max_{j=1}^{n}(w_{i,j}, h_{i,j} \times g) + b \qquad \begin{array}{l} \text{for } n \text{ local computations in superstep } i, \text{ and} \\ h_{i,j} \text{ is } h\text{-relation of local computation } j \text{ in superstep } i \end{array} \tag{4.3}$$

4.3 Reliable Grid Computing

In a grid, a large number of computational and data resources are dynamically coordinated to execute large scale projects involving collaborating scientists, high-performance computers, massive data stores, and large scale scientific instruments. Grids such as those run by CERN

and other flagship research environments today routinely process terabyte to petabyte-scale event data. As grid computing enters the mainstream and is applied in internet search, finance, large-scale engineering design and other domains, key issues like interoperability, security, and fault tolerance grow in importance. Since the objective of this research is to improve the reliability of grid applications, further discussions are limited to *fault tolerance*.

4.3.1 The Need for Fault Tolerance

Failure in grids is arguably inevitable due to

- the heterogeneity and the massive scale of grid resources,
- the distribution of such resources over unreliable networks,
- the complexity of mechanisms that integrate these resources into a seamless utility, and
- the dynamic nature of the grid infrastructure which allows continuous changes to happen.

Hundreds and thousands of machines are coordinated to execute a grid application. Even when a grid infrastructure is composed of highly reliable resources, which is not always the case, due to the sheer scale of grid resources, it is highly likely for some of these resources to fail while a computation is in progress.

Grid resources that are exchanging a bulk of data should be able to establish and maintain a network session for an extended period of time. For this, many network components are involved in routing data from the source to destination. Due to the size of the transferred data and the involvement of many network components, failure during data transmission does not come as a surprise in grids.

The core grid services play a vital role in enabling highly secure and dynamic sharing of thousands of heterogeneous resources among multiple collaborators. However, the complexity of the responsibilities of such services, coupled with the services not being 100% reliable, is yet another reason for a failure to occur in grids.

In grids, constant change is the norm rather than the exception. Grid resources may join and leave the resource pool of a virtual organization at any time, or an administrative domain may change its security policy without notifying any concerned parties. If any of these or other similar events happen during the execution of a grid application, the execution fails. For example, if a resource which is assigned to execute an activity of a grid application suddenly disappears from the resource pool, the application will be waiting forever to get the output of

that activity. Similarly, if access to specific files is blocked during execution or a major software upgrade is performed that might cause incompatibility with the current execution, then the execution will fail.

The cause of failure in grids comes not only from grid resources and services but also from grid applications themselves. Grid applications are often long running. Since the activities of a grid application and their execution environment are not necessarily fault free, the longevity of a grid application execution increases the chance for some of the faults to cause errors before the execution is completed.

Multiple factors almost make the occurrence of a failure during the execution of a grid application a practical certainty. Therefore, fault tolerance support is needed to guarantee the successful completion of a grid application execution in the face of many threats.

4.3.2 The State-of-the-Art

The grid community has proposed various fault tolerance approaches, which are based on traditional fault tolerance techniques (Section 3.3), to increase the probability of successful execution of grid applications. Many of these approaches are focused on either improving the reliability of a single grid application or the reliability of multiple grid applications in a workflow. Therefore, we broadly classify fault tolerance solutions for grids into two categories: *application/service-based* and *workflow-based*. In Sections 4.3.2.1 and 4.3.2.2, we discuss the the general direction that is taken to provide fault tolerance support to grid applications in each category.

4.3.2.1 Application/Service-Based Fault Tolerance Solutions

Checkpointing and replication are popular fault tolerance strategies that are used extensively for providing fault tolerance support to grid applications and services. These strategies are used as foundations of many grid-based fault tolerance researches [Silva et al., 2003, Budati et al., 2007, Nazir et al., 2009, Chtepen et al., 2009]. The difference between these researches mainly comes from how each research exploits the parameters of the fault tolerance strategies, such as checkpointing interval and the number of replicas, to achieve highly reliable computation. Other fault tolerance strategies like N-version [Xu et al., 2008] and restart [Dean and Ghemawat, 2004] are also used.

Nazir et al. [2009] and Chtepen et al. [2009] proposed adaptive checkpointing to manage the failure of activities during the execution of a grid application. Both researches, though in a

different manner, use the performance of grid resources to adjust checkpointing interval. Nazir et al. [2009] evaluates the performance of a resource based on how often the resource completes the execution of an activity within the deadline. If a resource is known to be prone to failure, i.e., the resource does not complete activity execution within the deadline more often than not, then the activities that are executed on the resource will be checkpointed frequently, and vice versa.

Chtepen et al. [2009], on the other hand, combines the failure frequency of a resource with the execution time of an activity to adjust checkpointing interval or determine the need to checkpoint at runtime. If the resource is either stable or the activity is about to complete, then the checkpointing interval is increased to minimize checkpointing frequency, and vice versa. On occasions when the checkpointing interval is fixed, activities are checkpointed only if their execution environment is considered to be unstable. Otherwise, the checkpointing is skipped. Chtepen et al. [2009] also proposed adaptive replication. The execution of the replica of an activity is determined based on the load of the grid infrastructure. If resources are not available to execute all replicas of an activity, for instance during peak hours, then the execution of some or all of the replicas will be postponed. .

Earlier works, such as by Li and Mascagni [2003], Silva et al. [2003] and Budati et al. [2007], also proposed replication-based fault tolerance approaches for grids. Li and Mascagni [2003] assumed computational grid resources to be unlimited, and thus they proposed replicating each activity of a grid application n times irrespective of the current status of the execution environment. However, they acknowledged that using arbitrarily large number of replicas would not necessarily lead to a better reliability, and the execution of unnecessary replicas would significantly increase system workload. Therefore, they proposed an analytical model, which takes into account the performance and the availability of grid resources, to determine the minimum number of replicas that could satisfy performance requirements.

The replication-based fault tolerance support by Silva et al. [2003], unlike by Li and Mascagni [2003], does not assume 'infinite' grid resources. Once all activities of a grid application are allocated an execution environment, each activity is replicated up to a threshold *only if* there are idle resources to execute the replicas. This replication approach is similar to the one proposed by Chtepen et al. [2009], the difference being Chtepen et al. [2009] do not wait until all activities of the grid application are allocated an execution environment before replicating activities. All of the replication-based fault tolerance approaches that we have discussed, when one replica completes successfully, terminate the executions of all other replicas.

Chtepen et al. [2009], Li and Mascagni [2003] and Silva et al. [2003] use replication to guarantee the completion of the execution of an activity. Nevertheless, Budati et al. [2007] replicate activities not only to increase the chance of completion but also to ensure that the output of each execution is correct. For this, they combine replication with voting. The execution of an activity is completed when the required number of replicas reach consensus on the final result. In this work, the performance of resources is used to determine the number of required replicas that guarantee correct and timely execution of activities. The performance of each resource is evaluated by the correctness of the output of activities that the resource hosted, and the ability of the resource to complete execution within expected time frame.

Xu et al. [2008] took advantage of the opportunity that is presented by the Service-Oriented Architecture [Papazoglou and Heuvel, 2007] for dynamically locating multiple equivalent services, and proposed N-version with voting to manage failure of grid services. For this, they provide the FT-Grid service that searches multiple equivalent services to achieve a given goal. The FT-Grid service invokes services that are selected by its client, collects the outputs from these services, and finally returns the consensus result to the client. Other research work for managing failure of grid services include by Zhang et al. [2004], Jurgen and Thomas [2008] and Cesario and Talia [2011]. Zhang et al. [2004] and Cesario and Talia [2011] manage failure by primary-backup approach, while Jurgen and Thomas [2008] focus on tolerating the failure of a grid service due to a byzantine fault, a fault that causes the service to behave arbitrarily.

The last research work that we will discuss is concerned with MapReduce applications from Google Inc. Though Dean and Ghemawat [2004] did not explicitly state whether the MapReduce applications at Google are executed on a grid infrastructure or not, the work is highly relevant to our research due to its focus on one of the dwarfs that we will study later in detail. According to the discussion by Dean and Ghemawat [2004], the failure of map and reduce activities are tolerated mostly by restart and on occasion by replication. If the execution environment of map activities fails, all map activities that are allocated to that resource will be re-executed. Even though some of the map activities may complete before the failure of their execution environment, they will be re-executed. This is because the output of each map activity is kept in a local disk, which becomes inaccessible when the resource fails. If the execution environment of reduce activities fails, on the other hand, only the ones that have not completed will be restarted. This is because the output of reduce activities are kept in a global file system. When the majority of the activities are completed, the rest, also knows as the *stragglers*, are replicated to speed up their completion. Though the primary purpose of the replication is not to tolerate failure, it serves a double purpose especially if some of the

stragglers fail.

4.3.2.2 Workflow-Based Fault Tolerance Solutions

Workflow-based fault tolerance solutions aim to improve the reliability of a grid workflow execution. Research works by, for example, Hwang and Kesselman [2003], Kandaswamy et al. [2008] and Zhang et al. [2009] focus on tolerating task failures at the workflow and task levels. There are also workflow management systems, such as DAGMan, ASKALON, Taverna and Kepler, that provide fault tolerance support to grid workflows. Below, we discuss representative research works and workflow management systems that give the overall picture about workflow-based fault tolerance handling in grids.

The Grid Workflow System framework (Grid-WFS) was proposed by Hwang and Kesselman [2003] for handling task failures during the execution of acyclic grid workflows. Grid-WFS provides fault tolerance support using restart, checkpointing, replication, standby spare, N-version and the combination of these strategies (e.g., replication with restart). The framework also enables users to define what a task failure is in their workflow, which is referred to as *user-defined exception*, and how to handle that failure.

Zhang et al. [2009] proposed combining existing scheduling algorithms with replication and checkpointing to tolerate task failures in an acyclic grid workflow. In this work, tasks on which other tasks depend are replicated multiple times. The number of replica of each task is determined by performance and reliability constraints. Upon the completion of a task execution, the output of the execution is checkpointed. Zhang et al. [2009] also proposed replicating the entire workflow onto several clusters.

Kandaswamy et al. [2008] introduced a fault tolerance and recovery service (FTR) for improving the reliability of grid workflows using replication or migration. The decision of which fault tolerance strategy to use is based on parameters such as estimation of task execution time on a particular resource, expected queue and data transfer times, deadline and success constraints of the user, reliability models and availability of core grid services.

DAGMan [Couvares et al., 2007] uses a rescue file to tolerate failure of workflows. During the execution of a workflow, the status of each task, completed or failed, is recorded. In the event that the workflow fails, the user is able to recover the workflow using the rescue file. The rescue file ensures that only the tasks that are failed will be re-executed. Successfully completed tasks will not be re-executed. DAGMan can be configured in such a way that a failed task is repeatedly restarted up to a threshold before the status of the task is recorded as

failed.

ASKALON [Duan et al., 2005] checkpoints the state of the workflow as a whole and all intermediate data. Therefore, the executions of all tasks are suspended when a workflow is checkpointed. ASKALON checkpoints a workflow at predefined events, for example, when one of the tasks in the workflow fails or when a portion of the workflow execution is completed. In ASKALON, the failure of a task is tolerated by restart, while the failure of the entity that oversees the execution of the workflow is tolerated by redundancy.

Kepler [Crawl and Altintas, 2008] handles the failure of tasks during the execution of a workflow in similar manner as exception handling. If a task failure is detected, which is based on user-defined criteria, the recovery of the failed task is attempted either locally or at a higher level. Kepler uses restart and standby spare strategies to manage the failure of tasks. Kepler repeatedly attempts to restart the failed task or initiate the execution of the substitute task up to a threshold.

Similar to Kepler, Taverna [Oinn et al., 2006] uses restart and standby spare strategies to manage failure during a workflow execution. Taverna uses multiple restarts, with increasing delay intervals between restarts, to recover a failed task. If an alternative task is available, then the failed task will be substituted by its alternative. In Taverna, alternative tasks can be supplied statically before the execution of the workflow commences or dynamically while the execution is in progress.

Overall, workflow-based fault tolerance solutions, whether they are proposed by researchers or they are already part of workflow management systems, mainly apply fault tolerance strategies at the task level. Even the ones that provide workflow-level fault tolerance support also handle failures at the task level. For instance, ASKALON checkpoints at the workflow-level but restarts at the task level.

4.3.3 The Gap

Existing fault tolerance solutions offer various techniques for managing failure in grid applications, and services and workflows. However, these solutions lack one or more of the following features:

- *Architecture consideration.* Architecture-based fault tolerance support for grid application is not a well-studied subject. Despite the existence of numerous research effort for providing fault tolerance support in grids, we observe the lack of exploitation of the architecture of the application to improve reliability. Existing fault tolerance approaches are

either do not address the type of the grid application for which they are providing fault tolerance support (e.g., [Nazir et al., 2009]), or explicitly proposed for grid applications whose activities are embarrassingly parallel (e.g., [Chtepen et al., 2009]).

- *Proactive fault tolerance support.* Proactive fault tolerance strategies minimize the impact of the failure of an activity on the overall computation. They are especially useful if the failure occurs towards the end of a long running computation. Restarting such computation could be very costly. Although the fault tolerance support that is discussed by Dean and Ghemawat [2004] considers the architecture of the application, (i.e., MapReduce), failure of a map or reduce activity is in large part managed by restart.

- *Runtime prediction.* In cases where proactive fault tolerance strategies are used, existing approaches do not assess the *current* status of the computation or the *current* likelihood of failure in the execution environment at *runtime* before executing a proactive strategy. Instead, the history of grid resources and the load of the system are used in some of the approaches. Multiple copies of an activity are simultaneously executed, for instance, irrespective of the current status of the computation and the execution environment [Hwang and Kesselman, 2003], if the execution environment is *known* to be unreliable [Budati et al., 2007] or if the load of the environment is *low* [Silva et al., 2003].

- *Activity-level fault tolerance support.* This is commonly observed among workflow-based fault tolerance solutions. Recall that the tasks in a grid workflow represent grid applications or services (Section 2.3). Grid applications are naturally composed of multiple activities. Therefore, the cost of applying a fault tolerance strategy at the task level could be very high depending on the number of activities of the grid application, which is represented by the task, the execution time of each activity, and also the size of the intermediate output of each activity. In our workflow example in Section 2.3, restarting the image rendering task is potentially very expensive, especially if the failure occurs after most of the images are rendered. Since the overhead of workflow-level checkpointing significantly increases with the size of the intermediate output of a workflow execution [Duan et al., 2005], checkpointing the entire computation, as in the case in ASKALON, may introduce an unacceptably high cost.

4.3.4 The Solution

Each fault tolerance solution, whether application/service based or workflow based, plays a significant role in making grids reliable computing environment. However, in light of our discussion about the limitations of existing fault tolerance approaches in grids, a new complementary approach that addresses these limitations, if possible fully otherwise partially, is needed:

i. The new approach should systematically consider the *architecture* of grid applications. Since there are thousands of grid applications, it is tedious, if not impossible, to consider the architecture of all grid applications. Therefore, as a compromise, the *class* of a grid application should be taken into account to provide architecture-specific fault tolerance support. The class of a grid application can be determined by examining its communication and computation pattern. Luckily, based on such criterion, we will have only thirteen classes of grid applications (Section 4.1).

ii. The new approach should use both *proactive* and reactive strategies, as needed. Furthermore, proactive fault tolerance strategies should be executed based on *runtime* prediction. Other than on occasions when correctness validation is needed [Budati et al., 2007], executing multiple replica of an activity will most certainly waste resources. Though some claims about grid resources being infinite, for example by Li and Mascagni [2003], this is simply not the case. Resources are limited *and* some type of cost is associated with their usage. Therefore, the fault tolerance approach should make all possible effort to limit unnecessary resource utilisation. One way of doing this is, instead of using fixed number of replicas, an activity should be replicated only if the failure of the activity is imminent. Similar approach is taken by Chtepen et al. [2009] to dynamically omit unnecessary checkpointing.

iii. The new approach should execute a fault tolerance strategy at the *activity level*. As discussed in Section 4.3.3, the workflow-based solutions are penalized with respect to performance and cost for applying a fault tolerance strategy at the workflow-level, or at the task level provided that each task is composed of multiple activities. Even if a grid environment is prone to failure, it does not necessarily mean that significantly many activities will fail. Therefore, the target should be to limit recovery actions to failed activities, and proactive actions to activities that are likely to fail.

Using the above three points as guidelines, we propose a novel fault tolerance approach for grid applications. We refer to this approach as *the Recovery-Aware Components* approach.

4.4 Summary

This chapter presents the background theory that is related to design and reliable grid computing. We first defined software architecture. We discussed architectural styles, and then described four architectural styles in detail. For each architectural style, we presented the computational model, components, connectors, invariants, advantages, disadvantage and examples. Following this, we introduced the thirteen dwarfs, highlighted the unique features of each dwarf, and presented the Bulk Synchronous Parallel (BSP) model. In Section 4.3, the last section of this chapter, we discussed the need for fault tolerance support in grids, existing fault tolerance approaches for grids, the limitations of these approaches, and how these limitations could be addressed. Finally, we briefly introduced our novel fault tolerance approach for grids, the *Recovery-Aware Components* approach.

Part II

Recovery-Aware Component-Based Architecture

"I had a mother who taught me there is no such thing as failure.
It is just a temporary postponement of success."
- Buddy Ebsen

In this Part, we propose a novel fault tolerance approach, known as the Recovery-Aware Components approach, for grids. The Recovery-Aware Components approach is an architecture-based fault tolerance approach that is concerned with managing failure in a grid application execution reactively, and proactively—based on runtime prediction, at the activity level. In Chapter 5, we discuss how the different aspects of the approach are realized in a grid, and formally model the behaviour of a fault tolerant grid system that realizes the approach. In Chapter 6, we study how to manipulate the structural, computational and communicational pattern of a grid application to provide higher reliability improvement. Of the thirteen dwarfs, we study the customization of the Recovery-Aware Components approach for grid applications whose architecture is classified as either MapReduce dwarf or Combinational Logic dwarf.

Chapter 5

Recovery-Aware Components

The *Recovery-Aware Components (RAC) approach* is an architecture-based fault tolerance approach that is concerned with managing failure in a grid application execution reactively, and proactively—based on runtime prediction, at the activity level. The RAC approach is influenced by the principle of Recovery-Oriented Computing, which increases system availability and decreases cost of system ownership by decreasing the recovery time of failed systems (Section 3.5). The RAC approach aims to improve the reliability of a grid application execution and decrease the overhead of fault tolerance support by managing failure reactively and proactively at the smallest execution unit of the grid application.

The fundamental principles of the RAC approach are as follows.

i. Faults are identified and impending failures are predicted at *runtime*.

ii. The impact of the impending failures on the currently running grid application is either averted or minimized by *proactive* fault tolerance strategies.

iii. Where impending failures are not predicted or the failure aversion attempt is unsuccessful, failed activities are recovered by *reactive* fault tolerance strategies.

iv. Both reactive and proactive strategies are executed at the *activity level*: recovery actions are limited to failed activities while proactive actions are limited to activities that are predicted to fail.

v. The class of a grid application's *architecture* is taken into consideration, whenever possible, to provide customized fault tolerance support.

Runtime prediction plays an important role to provide a cost effective fault tolerance support to grid applications. In the RAC approach, a proactive fault tolerance strategy is executed only if the failure of activities is predicted to be imminent. By limiting the frequency a proactive strategy execution, we expect the overhead of the fault tolerance support to decrease without compromising reliability.

During a grid application execution, the hosts of the activities of the grid application are monitored for possible failure. If a host is predicted to fail, then a proactive strategy, such as checkpointing, migration and replication, is executed to tolerate the failure of the activities on that host. It is important to note that neither predicting *all* impending failures nor successfully averting the predicted ones *all* the time is a realistic expectation. Despite the pairing of runtime prediction with a proactive strategy, there is a non-zero probability for some activities to fail. On such occasions, failed activities are recovered by a reactive strategy like restart.

On occasions when a proactive strategy is successfully executed, the impact of the impending failure is either fully avoided or minimized. For example, if the activities are either migrated to or replicated on a different host, then the impending failure is fully averted. On the other hand, if the activities are checkpointed, the activities will fail along with their host. However, since checkpointed activities are rolled-back to their last stable state, and *not* started from the beginning, the impact of the failure is minimized due to the checkpointing strategy.

The RAC approach provides cost-effective fault tolerance support to grid applications by not only pairing proactive strategies with runtime prediction but also managing failure at the activity level. Handling failures at the activity level presents an opportunity to confine failure locality, extent of recovery actions and thus fault tolerance overheads at user-defined granularity. Despite grids being failure prone execution environment, it does not necessarily mean that significantly many activities of a grid application will fail. Therefore, in the RAC approach, recovery actions are confined to failed activities, and proactive actions encompass only activities that are vulnerable to failure. We refer to a component that encapsulates the functionality of an activity of a grid application *and* has an interface through which fault tolerance support can be projected to the activity as a *recovery-aware component*. Fault tolerance managers monitor and interact with activities of grid applications via component interfaces. A grid application that is composed of recovery-aware components is referred to as a *RAC-based grid application*.

The RAC approach also provides customized fault tolerance support by manipulating the structural, computational and communicational pattern of grid applications. Further discussion about customized fault tolerance support is available in Chapter 6.

5.1 Recovery-Aware Component-Based System

A *Recovery-Aware Component-Based System (RACS)* is a fault tolerant grid system that realizes the RAC approach. A RACS provides fault tolerance support to RAC-based grid applications. It also provides an experiment testbed for evaluating the reliability of RAC-based grid applications. Figure 5.1 shows a UML [Booch et al., 2005] component diagram of a RACS reference architecture, which describes not only the structural relationship between the components of a RACS but also their mappings into a grid infrastructure. The components of a RACS are *Head Manager*, *Compute Manager*, *Predictor*, *Injector*, and *Recovery-aware component*. The roles of these components, their interactions, and their deployment on a grid infrastructure are discussed in subsequent sections.

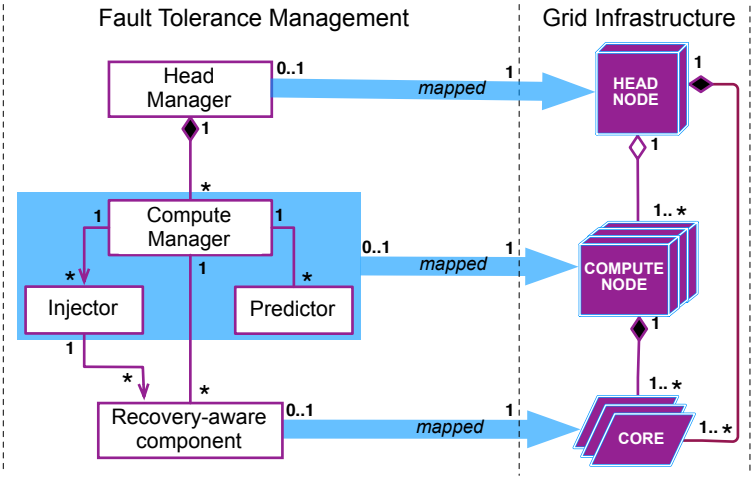


Figure 5.1: RACS Reference Architecture

In the RACS reference architecture, a grid infrastructure is depicted as being an execution environment with one head node and multiple compute nodes. By representing a grid infrastructure in this way, we are implying neither the compute nodes are under the management of the head node nor the compute nodes are homogeneous. The *head node* represents an entity, such as Xgrid controller, that is in charge of allocating the required grid resources to the activities of a grid application. A *compute node* represents any computing grid resource.

5.1.1 Roles

The roles of the components of a RACS are as follows.

5.1.1.1 Recovery-aware component

A recovery-aware component, as introduced previously, is a normal grid application component that has an additional interface for controlling some aspects of its fault tolerance affairs.

5.1.1.2 Injector

An injector introduces simulated and real faults into recovery-aware components and their execution environment. The injector is activated if a RACS is to be used as a fault tolerance testbed. The scope of fault injection is limited by the required type of failure simulation. For simulating a host crash, for example, the injector kills all currently running recovery-aware components on a given compute node. On the other hand, a CPU failure in a multi-core compute node is simulated by randomly choosing and terminating a recovery-aware component that is being executed.

5.1.1.3 Predictor

A predictor assesses the health of the currently running grid application and its environment, and then forecasts impending failures. A prediction has four possible outcomes (the sum of the probability of the prediction outcomes is 1):

i. *True Negative*: Failure is not imminent and is predicted to be non-imminent.

ii. *False Negative*: Failure is imminent but is predicted to be non-imminent.

iii. *True Positive*: Failure is imminent and is predicted to be imminent.

iv. *False Positive*: Failure is not imminent but is predicted to be imminent.

5.1.1.4 Compute Manager

A compute manager is responsible for starting predictors and injectors, and deciding when and how to take action to recover from or proactively prevent failure. The compute manager sets the frequency of failure prediction and fault injection, and notifies injectors the scope of fault introduction (simulating node failure vs. CPU failure). When a predictor makes a positive failure prediction, the compute manager either warns affected recovery-aware components to take necessary action or executes a proactive fault tolerance strategy on their behalf. The compute manager expects regular health updates from the recovery-aware components that

are under its management. If some of the recovery-aware components fail to send health updates, the compute manager marks those components as failed and executes a reactive fault tolerance strategy.

5.1.1.5 Head Manager

The head manager is responsible for starting compute managers. It also prepares detailed fault tolerance policies based on which compute managers make fault tolerance decisions. A fault tolerance policy includes the types of reactive and proactive strategies to be executed, the frequency of prediction and heart beat monitoring, and other fault tolerance related instructions. These policies can either be provided during the configuration of the head manager, *system FT policy*, or the submission of a grid application for execution, *user FT policy*.

5.1.2 Interactions

The head manager is the starting point for activating all parties that are involved in making a grid application fault tolerant. There are different ways to start the head manager: the head node starts the head manager whenever there is a submission of a RAC-based grid application, or the application itself directly initializes the head manager. Either way, the head manager commences execution and starts compute managers. The head manager sends a fault tolerance policy to the compute managers during their initialization.

Once the initialization of all available compute managers is completed, the grid application execution starts. During execution, one might choose for compute managers to send infrequent health updates to the head manager. This way, the head manager will be able to track any failed compute managers. If a particular compute manager fails to submit its health update, for example, the head manager assumes the compute manager has failed and attempts to recover it.

Predictors send warning messages to their respective compute managers when they predict failure in the near future. Depending on how accurate each predictor is, predictors make false positive predictions from time to time. This may subsequently lead compute managers to execute a proactive strategy unnecessarily.

If the injection interval and scope are included in the fault tolerance policy, compute managers pass this information to injectors when they initialize their respective injectors. However, if the value for these parameters is not provided, injectors use the default values of injection interval and scope.

Recovery-aware components register with their local compute manager prior to executing their main task. If a recovery-aware component fails to register, its manager does not provide any fault tolerance related service to the component either in the event of failure prediction or failure of the component. Recovery-aware components send regular health updates to their local compute manager until their execution is successfully completed or terminated. If the execution is completed successfully, they notify their local compute manager the end of the computation so that the manager no longer inspects these recovery-aware components. If a registered recovery-aware component does not notify the completion of its computation, its manager will wrongly conclude the component has failed (since health updates are no longer sent once a computation is completed) and then execute a reactive FT strategy.

5.1.3 Deployment

The head manager resides only on the head node. The primary role of the head manager is to configure and initialize compute managers when a grid application is submitted to the head node (see Section 5.1.1.5). The head manager does not have an active role during the execution of the grid application except when compute managers are required to send health updates. Thus, we argue, if the head manager fails, using reactive fault tolerance strategies such as restart is an acceptable way of handling its failure. Nevertheless, if compute managers need to send health updates, one might choose to place a redundant head manager on the head node.

On each **compute node**, there is at most one compute manager, one predictor, and one injector. A compute manager can oversee recovery-aware components deployed on many compute nodes. Since it is not unusual for a grid application to be executed in more than one cluster, a compute manager could be assigned to look after all recovery-aware components that run on a specific cluster. We expect deploying a predictor on each compute node to be useful since this allows provision of proactive fault tolerance support to all recovery-aware components. Each available **core** is allocated a maximum of one recovery-aware component.

5.1.4 The RAC Approach in the Context of Grids

The RAC approach is specifically applicable to the grid context through its design for local failure management, which addresses issues relating to heterogeneity and dynamic change. The activities of a grid application are executed on distributed resources, which potentially reside in multiple organizations. Since one of the key features of a grid is *controlled sharing*, some

resources may be off-limit for any purpose, other than regular activity execution, to non-local grid users. Therefore, it becomes difficult to provide fault tolerance support to activities that are being executed on such resources. However, such type of problems are bypassed in the RAC approach. The RAC approach advocates local failure management. As shown on the reference RACS architecture, Figure 5.1, and also discussed in Section 5.1.1.4, the compute managers are the ones that will do the heavy lifting, and the expectation is that each organization will have at least one compute manager. These local managers are, of course, not subjected to the same condition as the external ones.

5.2 Formal RACS Models

We formally model the global behaviour of a RACS by a finite-state DTMC (see Section 3.4.1). The RACS model is a parameterised Markov model. The parameterisation of the RACS model is in line with parameterised contracts and protocols aiming at providing accurate analysis about reusable components for specific deployment environments [Reussner et al., 2002]. Therefore, the RACS model is used to predict the reliability improvement a grid application would gain by adapting the RAC architecture, under various execution scenarios at a higher architectural level. The model is also used to estimate the overhead of such reliability improvement.

The global behaviour of a RACS is defined by the behaviours of individual predictors and recovery-aware components. In our RACS model, we exclude the behaviour of the head manager, compute managers and injectors. This is because the roles of the head and the compute managers are limited to facilitating the fault tolerance support, and therefore they are idle for the large part of the computation. The compute manager, for example, initiates the execution of a proactive strategy. Of course, the initialisation takes time; however, we assume that the time that is needed for such initialisation is significantly less than the execution of the proactive strategy itself. Therefore, for the sake of simplicity, we do not include such state in the RACS model. We followed the same reasoning to exclude the head manager's initialisation of compute managers and preparation of a fault tolerance policy. As for the injectors, their main purpose is to aid experiments, and thus are not the functional part of a RACS.

The current state of all recovery-aware components and predictors together define the current global state of a RACS. An individual RAC or predictor is in exactly one state at any given time. A predictor has two states, either it is predicting an impending failure, *predict* state, or doing nothing, *idle* state. A RAC, on the other hand, has seven states: *compute*, *failed*, *react*, *tp avert*, *fp avert*, *success* and *fatal error*. When the execution of a RAC is in

progress, the RAC is said to be in *compute* state. If the execution ends successfully, then the RAC is said to be in *success* state. Otherwise, the RAC is considered to be in *failed* state. Whenever a RAC fails, a recovery attempt is made. While the recovery attempt is in progress, the RAC stays in *react* state. If the recovery is successful, then the RAC returns to compute state. Otherwise, the RAC is assumed to fail beyond recovery, and this is called *fatal error* state, hereafter referred to as *error* state. During the execution of a RAC, a positive prediction will make the RAC be in *tp avert* state if the prediction is true or *fp avert* state if the prediction is false.

In the RACS model, *failed* and *idle* states are omitted. A recovery attempt will eventually be made to recover a failed RAC. Therefore, in the model, this is equivalent to transiting from *failed* state to *react* state with the transition probability of 1. Since such transition will not affect the reliability analysis, *failed* state is removed. With regards to *idle* state, this state is indirectly represented in the RACS model. If the RACS model is in any state other than predicting state, then it is effectively in the *idle* state. Therefore, there no need to explicitly include the *idle* state in the RACS model.

5.2.1 Global States

A RACS is modelled based on a strict reliability assumption. The execution of a RAC-based grid application is successful, i.e., the RACS is in a *global success* state, only if all of the recovery-aware components in the application complete their execution. If all uncompleted recovery-aware components are computing, no predictions are being made, and there are no failed recovery-aware components, then the system is considered to be in a *global compute* state. If any of the recovery-aware components fails, the system is in a *global react* state. In the react state, an attempt is made to recover the failed recovery-aware component(s). If the recovery attempt is successful, then the system will once again be in the compute state. Otherwise, the system is considered as failed beyond recovery, and this behaviour is referred to as a *global error* state. If predictions are being made and there are no failed recovery-aware components, then the system is said to be in a *global predict* state. After the prediction, if a proactive fault tolerance strategy is executed due to true positive prediction, then the system is said to be in a *global tp avert* state. However, if the prediction is false positive, then the system is said to be in a *global fp avert* state.

The global behaviour of a RACS is formally defined below.

- For every recovery-aware component x in a RACS, the DTMC states for x are labelled

as follows: $x.e$ for error state, $x.s$ for success state, $x.c$ for compute state, $x.r$ for react state, $x.t$ for tp avert state, and $x.f$ for fp avert state.

- For every predictor y in a RACS, the DTMC state for y is labelled as $y.p$ for predict state.

- Suppose g be a RACS, the DTMC states for g are labelled as follows: $g.e$ for global error state, $g.s$ for global success state, $g.c$ for global compute state, $g.r$ for global react state, $g.t$ for global tp avert state, and $g.f$ for global fp avert state. Equation (5.1) defines the global state interpretation of g.

$$g = \begin{cases} g.e & \text{iff } \exists x\ x.e \\ g.r & \text{iff } \forall x\ \neg x.e \wedge \exists x\ x.r \\ g.t & \text{iff } \forall x\ \neg x.e \wedge \forall x\ \neg x.r \wedge \exists x\ x.t \\ g.f & \text{iff } \forall x\ \neg x.e \wedge \forall x\ \neg x.r \wedge \forall x\ \neg x.t \wedge \exists x\ x.f \\ g.p & \text{iff } \forall x\ \neg x.e \wedge \forall x\ \neg x.r \wedge \forall x\ \neg x.t \wedge \forall x\ \neg x.f \wedge \exists x\ y.p \\ g.c & \text{iff } \forall x\ \neg x.e \wedge \forall x\ \neg x.r \wedge \forall x\ \neg x.t \wedge \forall x\ \neg x.f \wedge \forall x\ \neg y.p \wedge \exists x\ x.c \\ g.s & \text{iff } \forall x\ x.s \end{cases} \tag{5.1}$$

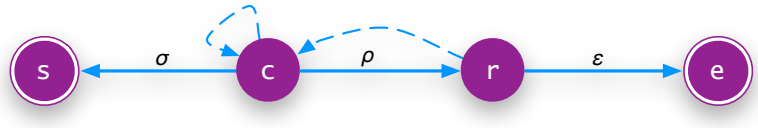
5.2.2 The Reactive RACS Model

We first introduce a simple RACS model in which failure is managed by only reactive fault tolerance strategies. We refer to this model as the *Reactive RACS model*. The reactive RACS model has only compute, react, error and success states. Therefore, the global state interpretation of a RACS in Equation (5.1) is updated by Equation (5.2).

$$g = \begin{cases} g.e & \text{iff } \exists x\ x.e \\ g.r & \text{iff } \forall x\ \neg x.e \wedge \exists x\ x.r \\ g.c & \text{iff } \forall x\ \neg x.e \wedge \forall x\ \neg x.r \wedge \exists x\ x.c \\ g.s & \text{iff } \forall x\ x.s \end{cases} \tag{5.2}$$

The global behaviour of a reactive RACS is modelled by a parameterised DTMC. The model is a matrix of rank 4, and is denoted by `GR`. Figure 5.2 and Equation (5.3) show the transition diagram and the transition matrix of `GR`, respectively. On the diagram, solid arrows represent a parameter entry in the underlying symbolic transition matrix. Non-absorbing states have exactly one outgoing dashed arrow, representing the symbolic probability (expression) enforcing that row sums of `GR` equal 1. The labels of the transition diagram map states and transitions

to GR. Each state label $i : a$ indicates its index $0 \leq i < 4$ in the transition matrix, and also an abbreviation a for the full state name: s for success, c for compute, r for react, and e for error.



Parameters $\sigma, \rho, \epsilon \in [0, 1]$

(i) → (j)	Description
———	$\mathtt{GR}(i, j)$ is a free variable
– – –	$\mathtt{GR}(i, j) = 1 - \sum_{k(k \neq j)} \mathtt{GR}(i, k)$

Figure 5.2: The parameterised DTMC of a reactive RACS (GR)

$$\begin{array}{c} \\ \mathbf{e} \\ \mathbf{s} \\ \mathbf{c} \\ \mathbf{r} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{e} & \mathbf{s} & \mathbf{c} & \mathbf{r} \end{array} \\ \begin{pmatrix} 1 & - & - & - \\ - & 1 & - & - \\ - & \sigma & 1 - \sigma - \rho & \rho \\ \epsilon & - & 1 - \epsilon & - \end{pmatrix} \end{array} \tag{5.3}$$

The parameters of GR are σ: probability of successful completion, ρ: probability of failure and ϵ: probability of unrecoverable failure, under the given constraint. To be more precise, the parameterised Markov model is $\mathtt{GR}(\sigma, \rho, \epsilon)$ and returns a concrete Markov model when a concrete combination of probabilities (constants) is substituted for the parameters. Thus, GR acts as a function from actual parameter values (within the parameter constraints specified) to concrete a Markov model (with constant probability matrices). For example in $\mathtt{GR}(0.0001, 0.00001, 0.001)$, the grid application has a low termination probability (i.e. long running) together with a five-nine reliability in its deployment resource context and a three-nine reliability of recovering from failure.

5.2.2.1 States and Transitions

The functional behaviour of a RAC-based grid application can be considered as a refinement state machine of compute. With probability σ, the execution of the application completes successfully in state success. A failure occurs during the execution with probability ρ, and

this causes the transition of GR from compute to react. If the recovery attempt in react is successful, then GR returns to compute. Otherwise, with probability ϵ, GR terminates with a catastrophic failure in error.

If a failure occurs during execution, we assume that the fault tolerance management always gets control on failure and attempts recovery. Hence, there is no direct transition from compute to error. In a sequential setting, this is not always realistic. A hardware failure, for example, could end the entire computation including the ability to manage faults. However, for a distributed grid application, without loss of generality, we assume that failures are recognised, and at least a recovery attempt is possible, such as bypassing faulty hardware or software.

5.2.2.2 Reliability Prediction

We use GR to predict the reliability of a RAC-based grid application to which a reactive RACS provides fault tolerance support. The properties of GR are given below:

- compute and react are transient states.
- compute is the entry and the exit state.
- success and error are absorbing states.
- The reliability of the grid application is the transition probability of GR from compute to success in the underlying absorption probability matrix.

We used MATLAB [MathWorks Website] to symbolically derive the absorption probability matrix of GR. Equation (5.4) shows the reliability of a reactive RACS, denoted by $RelR$.

$$RelR = \frac{\sigma}{\sigma + \epsilon \times \rho} \qquad \text{for } (\sigma + \epsilon \times \rho) > 0 \tag{5.4}$$

5.2.2.3 Overhead

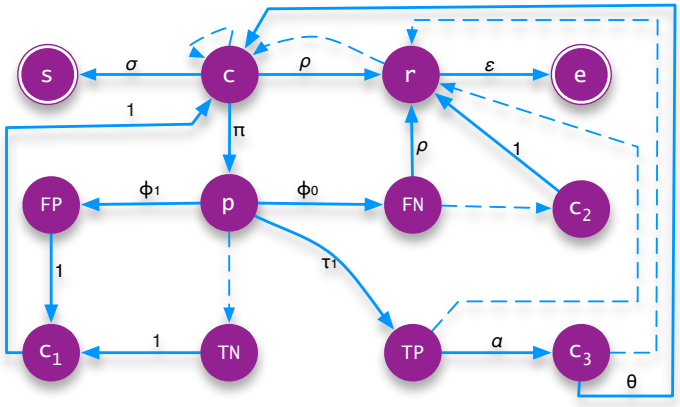
We use GR to estimate the overhead of the fault tolerance management of a reactive RACS. In the reactive RACS, the overhead comes from executing a reactive strategy. Therefore, the overhead of a reactive RACS is the product of the total number of times GR visits react during the course of the grid application execution, and the overhead of a single visit to react. The total number of visits to react is the transition of GR from compute to react in its potential matrix.

Let PR be the potential matrix of GR, and $O.\mathtt{r}$ be the overhead of a single visit to react. Equation (5.5) shows the overhead of a reactive RACS, denoted by OR.

$$\mathtt{OR} = \mathtt{PR}(\mathtt{c},\mathtt{r}) \times O.\mathtt{r} \tag{5.5}$$

5.2.3 The RACS Model

The global behaviour of a RACS is modelled by a parameterised DTMC. The model is a matrix of rank 12, and is denoted by G. G includes all states of GR and additional states that represent the proactive extension of GR: p for predict, TN for true negative, FN for false negative, TP for true positive, and FP for false positive; and intermediate computations: $\mathtt{c}_1$ for $\mathtt{compute}_1$, $\mathtt{c}_2$ for $\mathtt{compute}_2$, and $\mathtt{c}_3$ for $\mathtt{compute}_3$. Figure 5.3 and Equation (5.6) show the transition diagram and the transition matrix of G, respectively.



Parameters $\sigma, \rho, \epsilon, \pi, \tau_1, \phi_1, \phi_0, \alpha, \theta \in [0, 1]$

(i) → (j)	**Description**
———	$\mathtt{G}(i,j)$ is a free variable
— —	$\mathtt{G}(i,j) = 1 - \sum_{k(k\neq j)} \mathtt{G}(i,k)$

Figure 5.3: The parameterised DTMC of a RACS (G)

The parameters of G extend those of GR by π: the probability of prediction, τ_1: the probability of true positive prediction, ϕ_1: the probability of false positive prediction, ϕ_0: the probability of false negative prediction, α: the probability of the impending failure not occurring while proactive strategy execution is in progress, and θ: the probability of successful failure aversion.

$$
\begin{array}{c}
 \\
\mathbf{e} \\
\mathbf{s} \\
\mathbf{c} \\
\mathbf{r} \\
\mathbf{p} \\
\mathbf{TN} \\
\mathbf{FN} \\
\mathbf{TP} \\
\mathbf{FP} \\
\mathbf{c}_1 \\
\mathbf{c}_2 \\
\mathbf{c}_3
\end{array}
\begin{array}{c}
\begin{array}{ccccccccccc}
\mathbf{e} & \mathbf{s} & \mathbf{c} & \mathbf{r} & \mathbf{p} & \mathbf{TN} & \mathbf{FN} & \mathbf{TP} & \mathbf{FP} & \mathbf{c}_1 & \mathbf{c}_2 & \mathbf{c}_3
\end{array} \\
\left(\begin{array}{ccccccccccc}
1 & - & - & - & - & - & - & - & - & - & - & - \\
- & 1 & - & - & - & - & - & - & - & - & - & - \\
- & \sigma & 1\text{-}\sigma\text{-}\pi\text{-}\rho & \rho & \pi & - & - & - & - & - & - & - \\
\epsilon & - & 1\text{-}\epsilon & - & - & - & - & - & - & - & - & - \\
- & - & - & - & - & 1\text{-}\phi_0\text{-}\tau_1\text{-}\phi_1 & \phi_0 & \tau_1 & \phi_1 & - & - & - \\
- & - & - & - & - & - & - & - & - & 1 & - & - \\
- & - & - & \rho & - & - & - & - & - & - & 1\text{-}\rho & - \\
- & - & - & 1\text{-}\alpha & - & - & - & - & - & - & - & \alpha \\
- & - & - & - & - & - & - & - & - & 1 & - & - \\
- & - & 1 & - & - & - & - & - & - & - & - & - \\
- & - & - & 1 & - & - & - & - & - & - & - & - \\
- & - & \theta & 1\text{-}\theta & - & - & - & - & - & - & - & -
\end{array}\right)
\end{array}
\tag{5.6}
$$

5.2.3.1 States and Transitions

`success`, `compute`, `react` and `error` states are discussed in Section 5.2.2.1. A RACS makes prediction in `predict`. The prediction is either true negative, false negative, true positive or false positive (Section 5.1.1.3). After each respective prediction, `G` transits from `predict` to `true negative` with 1- ϕ_0-τ_1- ϕ_1 probability, `false negative` with ϕ_0 probability, `true positive` with τ_1 probability, and `false positive` with ϕ_1 probability.

In `G`, normal computation is represented by four states: `compute`, $\texttt{compute}_1$, $\texttt{compute}_2$, and $\texttt{compute}_3$. The need to add three more states to represent computation arises due to the impact of prediction outcomes on subsequent transitions and the memoryless property of Markov models. When `G` enters compute state for the first time after visiting `predict`, the next transition depends on the most recent prediction outcome. For example, if the prediction is false negative, the next transition after compute state is `react`. However, if the prediction is true negative, `G` stays in compute state. Since the next transition depends not only on the current state but also on a previous state, this behaviour violates the Markov property. The next transition should depend only on the current state, i.e. compute state, not on any previous state, i.e. any of the prediction outcomes states. Therefore, we add intermediate compute states to clearly show the subsequent transitions of `G` after visiting `predict` without violating the Markov property. We discuss how the transition proceeds after negative and positive predictions in Sections 5.2.3.1.1 5.2.3.1.2, respectively.

5.2.3.1.1 True and False Negative Predictions

When negative failure predictions are made, the fault tolerance management of a RACS will not intercept the computation of a grid application to execute a fault tolerance strategy. Therefore, no actual computation occurs in both `true negative` and `false negative` states. These states are there to show how the computation proceeds after either a true or a false negative prediction is made.

A true negative prediction confirms absence of failure before the next prediction. When a true negative prediction is made, `G` transits to $\texttt{compute}_1$with probability 1. Since there is no impending failure, `G` then transits to `compute` with probability 1, i.e., the system by definition will continue computing.

A false negative prediction confirms the presence of failure before the next prediction. When such prediction is made, the fault tolerance management of a RACS will not have a chance to initialise a proactive strategy execution. If the failure occurs right after the prediction, then `G` directly transits from `false negative` to `react`. Otherwise, `G` first transits to $\texttt{compute}_2$, i.e. the computation will continue for a while, and then when the computation fails, `G` transits from $\texttt{compute}_2$ to `react`. Either way, when a false negative prediction is made, `G` eventually transits to `react` with probability ϕ_0.

5.2.3.1.2 True and False Positive Predictions

Regardless of the correctness of the predictor, if a positive prediction is made, the fault tolerance management of a RACS intercepts the computation of a grid application and initialises a proactive strategy execution. The proactive strategy execution is carried out in either `true positive` for a correct prediction or `false positive` for an incorrect prediction.

After a true positive prediction, the impending failure will occur with $1 - \alpha$ probability while the proactive strategy is being executed—$\texttt{G}(\texttt{TP}, \texttt{r})$. If the proactive strategy execution is completed before the impending failure occurs—$\texttt{G}(\texttt{TP}, \texttt{c}_3)$, the impact of the impending failure on the overall computation will successfully be minimized or averted with θ probability.

Despite the absence of an impending failure, a false positive prediction causes a proactive strategy execution. Since there is no impending failure, after the completion of a proactive strategy, the transition probability distribution of `G` is the one after a true positive prediction. Therefore, after a proactive strategy is executed—$\texttt{G}(\texttt{FP}, \texttt{c}_3)$, the system continues computing—$\texttt{G}(\texttt{c}_3, \texttt{c})$.

5.2.3.2 The Simplified RACS Model

The RACS model provides a detailed information about what the behaviour of a RACS looks like, in particular after a prediction is made. However, since the main motivation for constructing the model is to analyse the reliability improvement a grid application would gain by adapting the RAC architecture and the overhead of such improvement, this level of detail is unnecessary. Therefore, some of the states can be removed without affecting the reliability and cost analyses, provided that their transition probabilities are preserved.

The simplified RACS model is constructed by removing `true negative`, `false negative`, $\texttt{compute}_1$, $\texttt{compute}_2$, and $\texttt{compute}_3$ from `G`. These states play an important role in clearly showing the possible transitions of `G` after a prediction. However, keeping the states in the model does not add any value for reliability and cost analysis as long as the transitions from and to these states are preserved. The preservation of the transition probabilities ensures `G` and its simplified model give identical reliability and cost analyses.

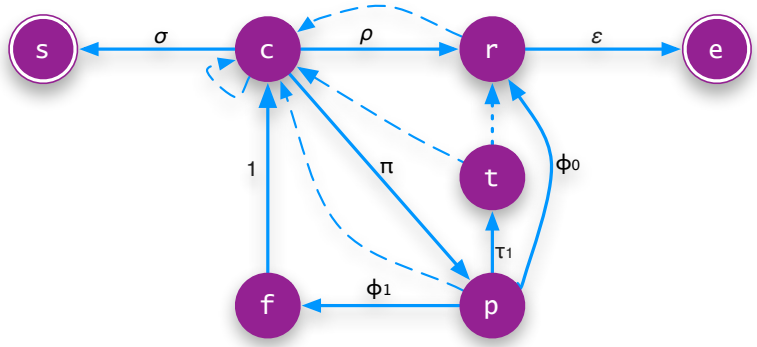
The simplified RACS model is a parameterised DTMC. The model is a matrix of rank 7, and is denoted by `GS`. *Note that* in order for their names to reflect the behaviour of the RACS when `GS` transits to `true positive` and `false positive`, the corresponding states are renamed as `tp avert` and `fp avert`, respectively. For the sake of uniformity, i.e. labelling all states with a single letter, the abbreviations of these states are changed from `TP` and `FP` to `t` and `f`, respectively. Figure 5.4 and Equation (5.7) show the transition diagram and the transition matrix of `GS`, respectively. In addition to the type of arrows in Figure 5.3, the transition diagram has a dotted arrow whose transition probability is expression in free variables. The states of `GS` are identical to the ones introduced in Section 5.2.1. For $\pi = 0$, `GS` and `GR` are identical.

5.2.3.2.1 Preserving Transition Probabilities

The transition probabilities of the removed states are preserved in `GS` as follows.

- Suppose a true negative prediction is made. Equation (5.8) shows the probability of `GS` to transit from `predict` to `compute`.

$$\begin{aligned} \texttt{GS}(\texttt{p},\texttt{c}) =& \texttt{G}(\texttt{p},\texttt{TN}) \times \texttt{G}(\texttt{TN},\texttt{c}_1) \times \texttt{G}(\texttt{c}_1,\texttt{c}) \\ =& (1-\phi_0-\tau_1-\phi_1) \times 1 \times 1 \end{aligned} \tag{5.8}$$



Parameters $\sigma, \rho, \epsilon, \pi, \tau_1, \phi_1, \phi_0, \alpha, \theta \in [0, 1]$

$(\mathbf{i}) \rightarrow (\mathbf{j})$	**Description**
———	$\texttt{GS}(i, j)$ is a free variable
— —	$\texttt{GS}(i, j) = 1 - \sum_{k(k \neq j)} \texttt{GS}(i, k)$
·····	$\texttt{GS}(i, j)$ is an expression in free variables

Figure 5.4: The parameterised DTMC of a simplified RACS (GS)

$$
\begin{array}{c} \\ \mathbf{e} \\ \mathbf{s} \\ \mathbf{c} \\ \mathbf{r} \\ \mathbf{p} \\ \mathbf{f} \\ \mathbf{t} \end{array}
\begin{array}{c}
\begin{array}{ccccccc} \mathbf{e} & \mathbf{s} & \mathbf{c} & \mathbf{r} & \mathbf{p} & \mathbf{f} & \mathbf{t} \end{array} \\
\begin{pmatrix}
1 & - & - & - & - & - & - \\
- & 1 & - & - & - & - & - \\
- & \sigma & 1\text{-}\sigma\text{-}\rho\text{-}\pi & \rho & \pi & - & - \\
\epsilon & - & 1\text{-}\epsilon & - & - & - & - \\
- & - & 1\text{-}\phi_0\text{-}\tau_1\text{-}\phi_1 & \phi_0 & - & \phi_1 & \tau_1 \\
- & - & 1 & - & - & - & - \\
- & - & \alpha\theta & 1-\alpha\theta & - & - & -
\end{pmatrix}
\end{array}
\tag{5.7}
$$

- Suppose a false negative prediction is made. Equation (5.9) shows the probability of GS to transit from `predict` to `react`.

$$
\begin{aligned}
\texttt{GS}(\texttt{p}, \texttt{r}) =& \texttt{G}(\texttt{p}, \texttt{FN}) \times (\texttt{G}(\texttt{FN}, \texttt{r}) + \texttt{G}(\texttt{FN}, \texttt{c}_2) \times \texttt{G}(\texttt{c}_2, \texttt{r})) \\
=& \phi_0 \times (\rho + ((1 - \rho) \times 1)) \\
=& \phi_0
\end{aligned}
\tag{5.9}
$$

- Suppose a true positive prediction is made. Equations (5.10) and (5.11) show the probability of GS to transit from `tp avert` to `compute` and `react`, respectively.

$$
\begin{aligned}
\texttt{GS}(\texttt{t}, \texttt{c}) =& \texttt{G}(\texttt{TP}, \texttt{c}_3) \times \texttt{G}(\texttt{c}_3, \texttt{c}) \\
=& \alpha \times \theta
\end{aligned}
\tag{5.10}
$$

$$\begin{aligned} \texttt{GS}(\texttt{t},\texttt{r}) =& \texttt{G}(\texttt{TP},\texttt{c}_3) \times \texttt{G}(\texttt{c}_3,\texttt{r}) + \texttt{G}(\texttt{TP},\texttt{r}) \\ =& \alpha \times (1-\theta) + (1-\alpha) \\ =& 1 - \theta \times \alpha \end{aligned} \tag{5.11}$$

- Suppose a false positive prediction is made. Equation (5.12) shows the probability of `GS` to transit from `fp avert` to `compute`.

$$\begin{aligned} \texttt{GS}(\texttt{f},\texttt{c}) =& \texttt{G}(\texttt{FP},\texttt{c}_1) \times \texttt{G}(\texttt{c}_1,\texttt{c}) \\ =& 1 \times 1 \end{aligned} \tag{5.12}$$

5.2.3.3 Reliability Prediction

We use `GS` to predict the reliability of a RAC-based grid application to which a RACS provides fault tolerance support. The properties of `GS` are given below:

- `compute`, `react`, `predict`, `fp avert` and `tp avert` are transient states.
- `compute` is the entry and the exit state.
- `success` and `error` are absorbing states.
- The reliability of the grid application is the transition probability of `GS` from `compute` to `success` in the underlying absorption probability matrix.

We used MATLAB [MathWorks Website] to symbolically derive the absorption probability matrix of `GS`. Equation (5.13) shows the reliability of a RACS, denoted by $RelH$.

$$\begin{aligned} RelH =& \frac{\sigma}{x} \\ \text{for } x =& \sigma + \epsilon \times \rho + \epsilon \times \phi_0 \times \pi + \epsilon \times \pi \times \tau_1 - \alpha \times \epsilon \times \pi \times \theta \times \tau_1, \\ x >& 0 \end{aligned} \tag{5.13}$$

5.2.3.4 Overhead

We use `GS` to estimate the overhead of the fault tolerance management of a RACS. In the RACS, the overhead comes from executing a reactive strategy, a proactive strategy and making predictions. Therefore, the overhead of a RACS is the weighted sum of the expected number of visits from `compute` to `react`, `predict`, `tp avert` and `fp avert`. The total number of visits to each of these states is weighted by the overhead of a single visit to each state. The expected number of visits are obtained from the potential matrix of `GS`.

Let PS be the potential matrix of GS, and $O.\mathtt{i}$ be the overhead of a single visit to state i. Equation (5.14) shows the overhead of a RACS, denoted by OS.

$$\begin{aligned}\mathtt{OS} = \mathtt{PS}(\mathtt{c},\mathtt{r}) \times O.\mathtt{r} + \mathtt{PS}(\mathtt{c},\mathtt{p}) \times O.\mathtt{p} \\ + \mathtt{PS}(\mathtt{c},\mathtt{t}) \times O.\mathtt{t} + \mathtt{PS}(\mathtt{c},\mathtt{f}) \times O.\mathtt{f}\end{aligned} \tag{5.14}$$

5.3 BSP-Based RACS Models

The previous RACS models represent the entire grid application execution by one state, i.e., compute. Here, we modify the reactive and the simplified RACS models to provide a high-level abstraction of a grid application execution according to the principles of Valiant's BSP model (Section 4.2). We assume a grid application as being a sequence of supersteps. Therefore, we refine compute into three states that represent local computation ($\mathtt{c_l}$), global communication ($\mathtt{c_g}$), and barrier synchronisation ($\mathtt{c_b}$).

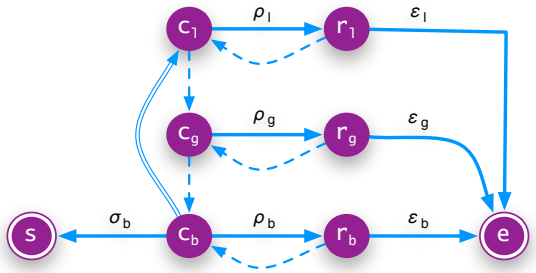
5.3.1 The Reactive RACS Model and BSP

We first refine the reactive RACS model (Section 5.2.2) based on the principles of the BSP model. We refer to the refined reactive RACS as the *BSP-based reactive RACS model*. The BSP-based reactive RACS model is a parameterised DTMC. The model is a matrix of rank 8, and is denoted by GRbsp. In GRbsp, each new compute state ($\mathtt{c_l}$, $\mathtt{c_g}$, and $\mathtt{c_b}$) has a corresponding react state ($\mathtt{r_l}$, $\mathtt{r_g}$, and $\mathtt{r_b}$). The parameters of GRbsp are $\sigma_{\mathtt{b}}$, $\rho_{\mathtt{l}}$, $\rho_{\mathtt{g}}$, $\rho_{\mathtt{b}}$, $\epsilon_{\mathtt{l}}$, $\epsilon_{\mathtt{g}}$, and $\epsilon_{\mathtt{b}}$. Figure 5.5 and Equation (5.15) show the transition diagram and the transition matrix of GRbsp, respectively.

5.3.1.1 States and Transitions

GRbsp transits from $\mathtt{c_l}$ to $\mathtt{r_l}$ with probability of $\rho_{\mathtt{l}}$ in the event of failure in the local computations, and from $\mathtt{c_l}$ to $\mathtt{c_g}$ when all local commutations successfully complete. Once in $\mathtt{c_g}$, GRbsp transits to $\mathtt{c_b}$ if all communications are carried out with success. Otherwise, with probability of $\rho_{\mathtt{g}}$, GRbsp transits from $\mathtt{c_g}$ to $\mathtt{r_g}$. After transiting to $\mathtt{c_b}$, there is $\rho_{\mathtt{b}}$ chance for GRbsp to transit to $\mathtt{r_b}$. Otherwise, GRbsp return to $\mathtt{c_l}$ to commence the next superstep. This continues until all supersteps are successfully completed. Upon successful completion of all supersteps, GRbsp transits from $\mathtt{c_b}$ to s with the probability of $\sigma_{\mathtt{b}}$. The transitions between $\mathtt{c_l}$ and $\mathtt{c_g}$, $\mathtt{c_g}$ and $\mathtt{c_b}$, and $\mathtt{c_b}$ and $\mathtt{c_l}$ represent the selfloop compute state transition of GR (see Figure 5.2).

For simplicity, we assume that the failure probability of each component of a superstep is similar in all supersteps. For instance, the probability of transiting from $\mathtt{c_l}$ to $\mathtt{r_l}$ is $\rho_{\mathtt{l}}$ in



(i) → (j)	Description
———	$\texttt{GRbsp}(i,j)$ is a free variable
– – –	$\texttt{GRbsp}(i,j) = 1 - \sum_{k(k\neq j)} \texttt{GRbsp}(i,k)$
=	a transition to the next superstep, and $\texttt{GRbsp}(i,j) = 1 - \sum_{k(k\neq j)} \texttt{GRbsp}(i,k)$

Figure 5.5: A BSP-based reactive RACS model (`GRbsp`)

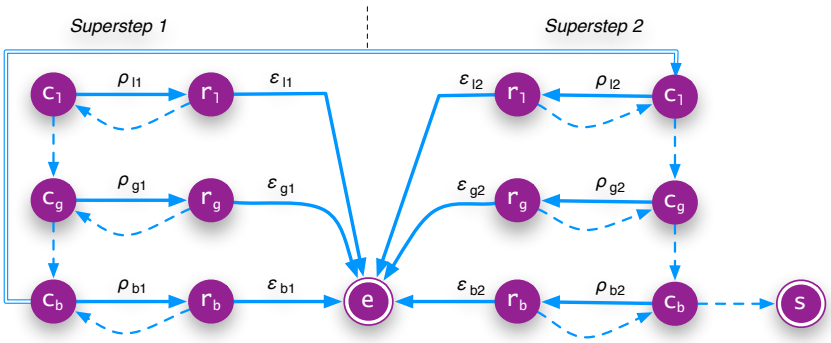
$$
\begin{array}{c}
\\
\mathbf{e} \\
\mathbf{s} \\
\mathbf{c_l} \\
\mathbf{c_g} \\
\mathbf{c_b} \\
\mathbf{r_l} \\
\mathbf{r_g} \\
\mathbf{r_b}
\end{array}
\begin{array}{c}
\begin{array}{cccccccc}
\mathbf{e} & \mathbf{s} & \mathbf{c_l} & \mathbf{c_g} & \mathbf{c_b} & \mathbf{r_l} & \mathbf{r_g} & \mathbf{r_b}
\end{array} \\
\begin{pmatrix}
1 & - & - & - & - & - & - & - \\
- & 1 & - & - & - & - & - & - \\
- & - & - & 1\text{-}\rho_l & - & \rho_l & - & - \\
- & - & - & - & 1-\rho_g & - & \rho_g & - \\
- & \sigma_b & 1\text{-}\rho_b\text{-}\sigma_b & - & - & - & - & \rho_b \\
\epsilon_l & - & 1\text{-}\epsilon_l & - & - & - & - & - \\
\epsilon_g & - & - & 1\text{-}\epsilon_g & - & - & - & - \\
\epsilon_b & - & - & - & 1\text{-}\epsilon_b & - & - & -
\end{pmatrix}
\end{array}
\tag{5.15}
$$

all supersteps. However, one might unfold the current model without difficulty and assign to the three components of a superstep different transition probabilities in each superstep. Figure 5.6 shows the BSP-based reactive RACS model of a grid application execution that has two supersteps. The model transits from $\texttt{c}_\texttt{l}$ to $\texttt{r}_\texttt{l}$ with the probability of ρ_{l1} and ρ_{l2} in supersteps 1 and 2, respectively.

5.3.1.2 Reliability Prediction

We use `GRbsp` to predict the reliability of a RAC-based grid application to which a reactive RACS provides fault tolerance support. The properties of `GRbsp` are given below:

- $\texttt{c}_\texttt{l}$, $\texttt{c}_\texttt{g}$, $\texttt{c}_\texttt{b}$, $\texttt{r}_\texttt{l}$, $\texttt{r}_\texttt{g}$ and $\texttt{r}_\texttt{b}$ are transient states.



(i) → (j)	Description
=	a transition to the next superstep, and $\texttt{GRbsp}(i,j) = 1 - \sum_{k(k \neq j)} \texttt{GRbsp}(i,k)$

Figure 5.6: Example: a BSP-based reactive RACS model with 2 supersteps

- $\texttt{c}_\texttt{l}$ is the entry state and $\texttt{c}_\texttt{b}$ is the exit state.
- `s` and `e` are absorbing states.
- The reliability of the grid application is the transition probability of `GRbsp` from $\texttt{c}_\texttt{l}$ to `s` in the underlying absorption probability matrix.

We used MATLAB [MathWorks Website] to symbolically derive the absorption probability matrix of `GRbsp`. Due to the size of the reliability equation, we do not show the equation here.

5.3.1.3 Overhead

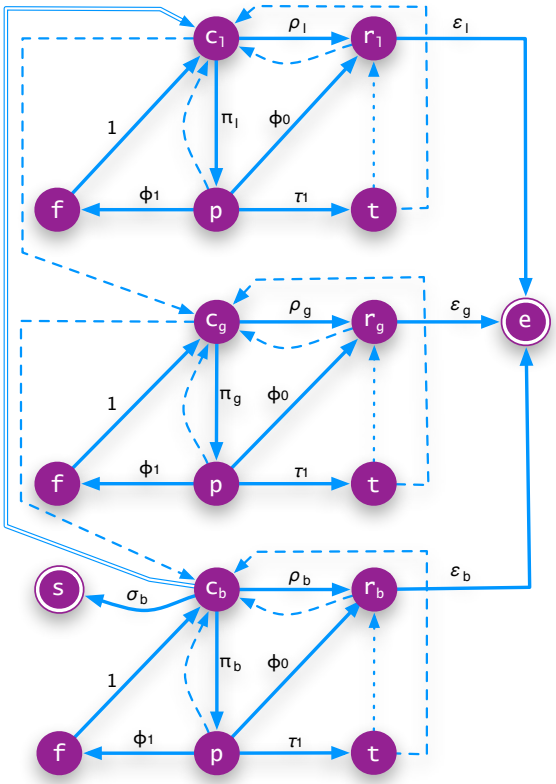
We use `GRbsp` to estimate the overhead of the fault tolerance management of a reactive RACS. In the reactive RACS, the overhead comes from executing a reactive strategy. Therefore, the overhead of a reactive RACS is the weighted sum of the expected number of visits from $\texttt{c}_\texttt{l}$ to $\texttt{r}_\texttt{l}$, $\texttt{r}_\texttt{g}$, and $\texttt{r}_\texttt{b}$. The total number of visits to each of these states is weighted by the overhead of a single visit to each state. The expected number of visits are obtained from the potential matrix of `GRbsp`.

Let `PRbsp` be the potential matrix of `GRbsp`, and $O.\texttt{i}$ be the overhead of a single visit to state `i`. Equation (5.16) shows the overhead of a RACS, denoted by `ORbsp`.

$$\begin{aligned}\texttt{ORbsp} = {} & \texttt{PRbsp}(\texttt{c}_\texttt{l}, \texttt{r}_\texttt{l}) \times O.\texttt{r}_\texttt{l} + \texttt{PRbsp}(\texttt{c}_\texttt{l}, \texttt{r}_\texttt{g}) \times O.\texttt{r}_\texttt{g} \\ & + \texttt{PRbsp}(\texttt{c}_\texttt{l}, \texttt{r}_\texttt{b}) \times O.\texttt{r}_\texttt{b}\end{aligned} \tag{5.16}$$

5.3.2 The RACS Model and BSP

We refine the RACS model (Section 5.2.3.2) based on the principles of the BSP model. The refined model is called *The BSP-based RACS model*. The BSP-based RACS model is a parameterised DTMC. The model is a matrix of rank 17, and is denoted by `Gbsp`. `Gbsp` extends `GRbsp` by `predict`, `tp avert` and `fp avert`. In `Gbsp`, each new `compute` state ($\mathtt{c}_\mathtt{l}$, $\mathtt{c}_\mathtt{g}$, and $\mathtt{c}_\mathtt{b}$) has a corresponding `react` state ($\mathtt{r}_\mathtt{l}$, $\mathtt{r}_\mathtt{g}$, and $\mathtt{r}_\mathtt{b}$), `predict` state ($\mathtt{p}_\mathtt{l}$, $\mathtt{p}_\mathtt{g}$ and $\mathtt{p}_\mathtt{b}$), `tp avert` ($\mathtt{t}_\mathtt{l}$, $\mathtt{t}_\mathtt{g}$ or $\mathtt{t}_\mathtt{b}$) state and `fp avert` state ($\mathtt{f}_\mathtt{l}$, $\mathtt{f}_\mathtt{g}$ and $\mathtt{f}_\mathtt{b}$). The parameters of `Gbsp` are $\sigma_\mathtt{b}$, $\rho_\mathtt{l}$, $\rho_\mathtt{g}$, $\rho_\mathtt{b}$, $\epsilon_\mathtt{l}$, $\epsilon_\mathtt{g}$, $\epsilon_\mathtt{b}$, $\pi_\mathtt{l}$, $\pi_\mathtt{g}$, $\pi_\mathtt{b}$, $\tau_\mathtt{l}$, $\phi_\mathtt{l}$, ϕ_0, α and θ. Figure 5.7 shows the transition matrix of `Gbsp`.



(i) → (j)	Description
=	a transition to the next superstep, and $\mathtt{Gbsp}(i,j) = 1 - \sum_{k(k\neq j)} \mathtt{Gbsp}(i,k)$

Figure 5.7: A BSP-based RACS model (`Gbsp`)

5.3.2.1 States and Transitions

When a prediction is made, Gbsp transits from the current compute state (c_l, c_g, or c_b) to corresponding predict state (p_l, p_g or p_b). If the prediction is correct, Gbsp transits from predict to respective tp avert state (t_l, t_g or t_b). Otherwise, Gbsp transits to respective fp avert state (f_l, f_g or f_b). The transitions between c_l and c_g, c_g and c_b, and c_b and c_l represent the selfloop compute state transition of GS (see Figure 5.4).

Gbsp is simplified by the following assumptions:

i. Proactive fault tolerance strategies are provided only for local computation. Thus, failure prediction is required only when Gbsp is in c_l.

ii. Failure in communication and barrier synchronisation is tolerated by only reactive fault tolerance strategies. Therefore, p_g, p_b, t_g, t_b, f_g, and f_b are removed.

iii. The failure probability of each component of a superstep is similar in all supersteps.

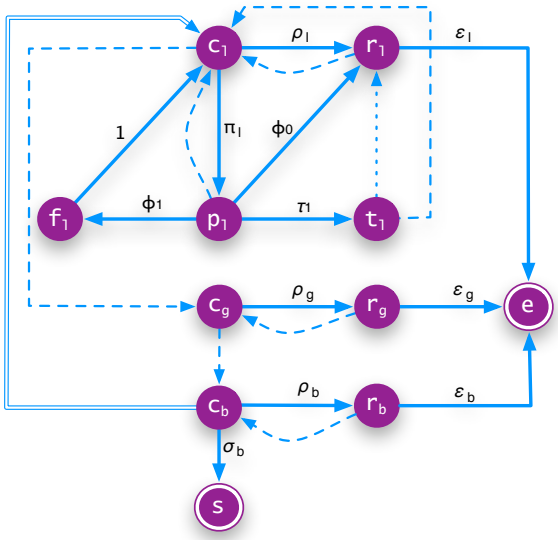
Figure 5.8 and Equation (5.17) show the transition diagram and the transition matrix of the simplified Gbsp, respectively. The simplified Gbsp is denoted by GSbsp. For $\pi_1 = 0$, GSbsp and GRbsp are identical.

5.3.2.2 Reliability Prediction

We use GSbsp to predict the reliability of a RAC-based grid application to which a RACS provides fault tolerance support. The properties of GSbsp are given below:

- c_l, r_l, p_l, f_l, t_l, c_g, r_g, c_b and r_b are transient states.
- c_l is the entry state and c_b is the exit state.
- s and e are absorbing states.
- The reliability of the grid application is the transition probability of GSbsp from c_l to s in the underlying absorption probability matrix.

We used MATLAB [MathWorks Website] to symbolically derive the absorption probability matrix of GSbsp. Due to the size of the reliability equation, we do not show the equation here.



(i) → (j)	Description
———	$\texttt{GSbsp}(i,j)$ is a free variable
— —	$\texttt{GSbsp}(i,j) = 1 - \sum_{k(k\neq j)} \texttt{GSbsp}(i,k)$
·····	$\texttt{GSbsp}(i,j)$ is an expression in free variables
=	a transition to the next superstep, $\texttt{GSbsp}(i,j) = 1 - \sum_{k(k\neq j)} \texttt{GSbsp}(i,k)$

Figure 5.8: A simplified BSP-based RACS model (GSbsp)

5.3.2.3 Overhead

We use GS to estimate the overhead of the fault tolerance management of a RACS. In the RACS, the overhead comes from executing a reactive strategy, a proactive strategy and making predictions. Therefore, the overhead of a RACS is the weighted sum of the expected number of visits from $\texttt{c}_\texttt{l}$ to $\texttt{r}_\texttt{l}$, $\texttt{r}_\texttt{g}$, $\texttt{r}_\texttt{b}$, $\texttt{f}_\texttt{l}$, $\texttt{t}_\texttt{l}$ and $\texttt{p}_\texttt{l}$. The total number of visits to each of these states is weighted by the overhead of a single visit to each state. The expected number of visits are obtained from the potential matrix of GSbsp.

Let PSbsp be the potential matrix of GSbsp, and $O.\texttt{i}$ be the overhead of a single visit to state i. Equation (5.18) shows the overhead of a RACS, denoted by OSbsp.

$$\begin{aligned}\texttt{OSbsp} = {} & \texttt{PSbsp}(\texttt{c}_\texttt{l},\texttt{r}_\texttt{l}) \times O.\texttt{r}_\texttt{l} + \texttt{PSbsp}(\texttt{c}_\texttt{l},\texttt{r}_\texttt{g}) \times O.\texttt{r}_\texttt{g} + \texttt{PSbsp}(\texttt{c}_\texttt{l},\texttt{r}_\texttt{b}) \times O.\texttt{r}_\texttt{b} \\ & + \texttt{PSbsp}(\texttt{c}_\texttt{l},\texttt{f}_\texttt{l}) \times O.\texttt{f}_\texttt{l} + \texttt{PSbsp}(\texttt{c}_\texttt{l},\texttt{t}_\texttt{l}) \times O.\texttt{t}_\texttt{l} + \texttt{PSbsp}(\texttt{c}_\texttt{l},\texttt{p}_\texttt{l}) \times O.\texttt{p}_\texttt{l}\end{aligned} \tag{5.18}$$

$$
\begin{array}{c}
 \\
\mathbf{e} \\
\mathbf{s} \\
\mathbf{c_l} \\
\mathbf{c_g} \\
\mathbf{c_b} \\
\mathbf{r_l} \\
\mathbf{r_g} \\
\mathbf{r_b} \\
\mathbf{p_l} \\
\mathbf{f_l} \\
\mathbf{t_l}
\end{array}
\begin{array}{c}
\begin{array}{cccccccccccc}
\mathbf{e} & \mathbf{s} & \mathbf{c_l} & \mathbf{c_g} & \mathbf{c_b} & \mathbf{r_l} & \mathbf{r_g} & \mathbf{r_b} & \mathbf{p_l} & \mathbf{f_l} & \mathbf{t_l}
\end{array} \\
\left(\begin{array}{cccccccccccc}
1 & - & - & - & - & - & - & - & - & - & - \\
- & 1 & - & - & - & - & - & - & - & - & - \\
- & - & - & 1\text{-}\rho_1\text{-}\pi_1 & - & \rho_1 & - & - & \pi_1 & - & - \\
- & - & - & - & 1\text{-}\rho_g & - & \rho_g & - & - & - & - \\
- & \sigma_b & 1\text{-}\sigma_b\text{-}\rho_b & - & - & - & - & \rho_b & - & - & - \\
\epsilon_1 & - & 1\text{-}\epsilon_1 & - & - & - & - & - & - & - & - \\
\epsilon_g & - & - & 1\text{-}\epsilon_g & - & - & - & - & - & - & - \\
\epsilon_b & - & - & - & 1\text{-}\epsilon_b & - & - & - & - & - & - \\
- & - & 1\text{-}\phi_0\text{-}\tau_1\text{-}\phi_1 & - & - & \phi_0 & - & - & - & \phi_1 & \tau_1 \\
- & - & 1 & - & - & - & - & - & - & - & - \\
- & - & \alpha\theta & - & - & 1\text{-}\alpha\theta & - & - & - & - & -
\end{array}\right)
\end{array}
\tag{5.17}
$$

5.4 Summary

In this chapter, we proposed the RAC approach. We then introduced a RACS, a fault tolerant grid system that realizes the RAC approach. We discussed the RACS based on its conceptual framework; we presented the roles of the framework's components, their interactions and their deployment in a grid infrastructure.

The second half of this chapter is focused on modelling the global behaviour of the RACS using Discrete Time Markov chains. We first modelled a RACS without prediction and proactive fault tolerance support. Then, we modelled the full behaviour of a RACS. We later refined these models based on the principles of Valiant's BSP model. For each model, we have shown how to compute the reliability of a grid application and the overhead of the fault tolerance management in a RACS.

Chapter 6

Architecture-Specific RAC

In Chapter 5, we proposed the RAC approach. We discussed all aspects of the RAC approach, except the exploitation of our knowledge about the behaviour of a grid application execution to provide better fault tolerance support, in detail. We refer to the RAC approach that does not take into account the architecture of a grid application as the *generic RAC approach*.

The generic RAC approach makes minimal assumptions about the architecture (activities and their interactions) of a grid application. The approach assumes the activities of a grid application to be the same for the purpose of fault tolerance management, and be embarrassingly parallel. The generic RAC based fault tolerance managers treat all activities equally irrespective of the degree of the impact of their failure on the overall computation. Furthermore, these managers attempt to recover a failed activity, except on certain circumstances, only if the activity is independent of all other activities. If the failed activity depends on previously completed activities, then the activity is usually considered as failed beyond recovery and no recovery attempt is made. The only exception when such activity can be recovered is if the failure of the activity is predicted and the state of the activity is successfully checkpointed. On such an occasion, the failed activity is rolled-back to the last saved state.

The generic RAC approach provides limited fault tolerance support to a grid application with activities that communicate, and/or in which the failure of each of its activities has variable impact on the overall computation. If there are dependencies between the activities of the grid application, an additional feature is needed to recover the failure of activities whose computation depends on other activities. Furthermore, since the failure of an activity on which other activities depend affects the overall computation more severely than an independent activity, activities that have severe impact on reliability should be given extra attention. Therefore, in order to address the issues in the generic RAC approach, i.e., recovering the

failure of dependent activities and providing additional care to guarantee the completion of important activities, one has to understand the interactions between the activities of a grid application and the significance of the role of each activity with respect to reliability.

Clearly, studying the architecture of *all* grid applications is impossible, due to, among other things, time constraints and the impracticality of locating all grid applications. Therefore, as a compromise, we consider the class of grid applications. For our study, we use the classification of parallel applications by Asanovic et al. [2006] to provide customized fault tolerance support to a *class* of grid applications rather than to a *specific* grid application. We refer to the RAC approach that manipulates the class of the architecture of grid applications to provide better fault tolerance support as the *architecture-specific RAC approach*.

Asanovic et al. [2006] classified parallel programs into thirteen computational kernels, known as *dwarfs* (Section 4.1). Each dwarf has a different kind of parallel coordination, i.e., communication and computation pattern. For each coordination, one can assume a different capability of utilising the parallel structure to increase reliability, and decrease cost of fault tolerance support. However, the actual reliability gain, cost reduction, and the constraints under which these can be achieved, if at all, are far from obvious and require some methodical approach and evaluation.

In this thesis, we study the reliability-overhead tradeoff that is enabled by the architecture-specific RAC approach for a grid application whose architecture can be classified under either the MapReduce (Section 6.1) or the Combinational Logic dwarf (Section 6.2). An analysis of all dwarfs was not feasible in the timeframe and so it was decided early on to focus on these dwarfs, which are both widely used and supported each on different open-source parallel platforms.

6.1 MapReduce Dwarf

The MapReduce (MR) dwarf represents parallel applications that are executed in two distinct phases; all execution units in the first phase are embarrassingly parallel, while the executions in the second phase involve some communication. Asanovic et al. [2006] defined the MR dwarf as "*... the essence is a single function that executes in parallel on independent data sets, with outputs that are eventually combined to form a single or small number of results.*"

The MR architecture is a well-known pattern in the functional programming paradigm [Field and Harrison, 1988, pp. 48−52]. This architecture is composed of two parameterised components: `map` and `reduce`. An idealized MR reference architecture is given in Figure 6.1.

The `map` component accepts a unary function m, and a collection of data type A and size d; where $a_0, a_1, \ldots a_{d-1} \in A$, and produces an intermediate collection of data type B; where $b_0, b_1, \ldots b_{d-1} \in B$. The `map` component applies m to each element a_i and outputs b_i. This is represented by level l in Figure 6.1. The signatures of m and `map` are shown below.

$$m : A \rightarrow B$$

$$\texttt{map} : m \ \# \ collection(A) \rightarrow collection(B)$$

The `reduce` component accepts an associative (possibly commutative) binary function ⓡ and collection of type B that was returned by `map`. The `reduce` component successively applies ⓡ to combine the collection into a single object C. This is represented by levels $[0, l\text{-}1]$ in Figure 6.1. The signatures of ⓡ and `reduce` are shown below, where the initial value of C is null.

$$ⓡ : B \ \# \ C \rightarrow C$$

$$\texttt{reduce} : ⓡ \ \# \ C \ \# \ collection(B) \rightarrow C$$

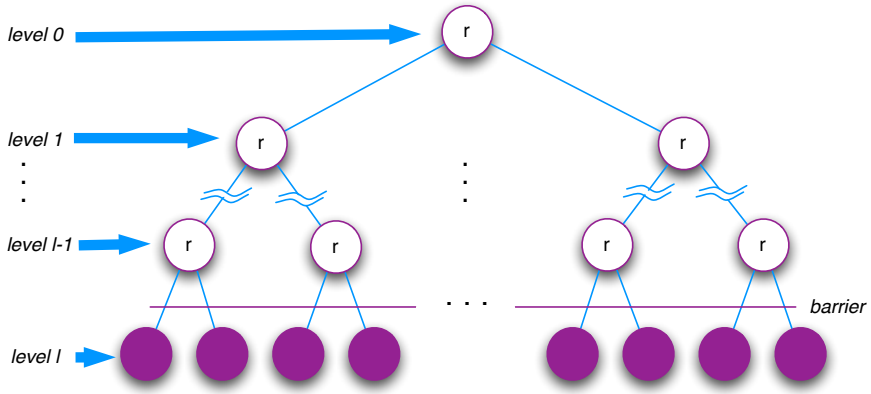


Figure 6.1: Idealized MapReduce Reference Architecture

Example 6.1 (Counting the occurrences of "jovial"). We use a simple example to illustrate how the `map` and `reduce` components work. In this example, we count the number of occurrences of the word "jovial" in a set of files.

- m scans a given file, and outputs the frequency of the word "jovial" in the file. Let F be a file, and N be the total number of "jovial" in F. The signature of m is shown below.

$$m : F \rightarrow N$$

Suppose the word "jovial" occurs four times in a file named *File1.txt*. Thus,

$$m : File1.txt \rightarrow 4$$

- The `map` component applies m to a set of files.

$$\texttt{map} : m \ \# \ collection(F) \rightarrow collection(N)$$

- After all files are scanned, the `reduce` component successively applies ⓡ on $collection(N)$ to get the final sum, denoted by T. When `reduce` applies ⓡ for the first time, the value of T is 0. The signature of `reduce` is shown below.

$$\texttt{reduce} : ⓡ \ \# \ T \ \# \ collection(N) \rightarrow T$$

- ⓡ accepts N and T. Then, ⓡ adds N and T and replaces T with the sum.

$$ⓡ : N \ \# \ T \rightarrow T$$

Suppose "jovial" occurs 11 times in the first i files, and 8 times in the $i + 1^{st}$ file. Thus,

$$ⓡ : 8 \ \# \ 11 \rightarrow 19$$

6.1.1 MapReduce Grid Applications

MapReduce grid applications are grid programs that are composed of `map` and `reduce` components. These applications are wide-spread and span a variety of domains such as machine learning, data mining, search, and image and video rendering. Some MR applications are discussed in Dean et al. [2008], Chen et al. [2009], Pantel et al. [2009], Karimzadehgan et al. [2011], GM et al. [2011], Suri and Vassilvitskii [2011], Chen et al. [2011], Rui Li et al. [2011] and Logofatu and Dumitrescu [2011]. Google's search is a notable example of an MR application [Dean and Ghemawat, 2004].

Recall, that components encapsulate activities, and these activities are protected against failure by fault tolerance mechanisms outside these components, but gracefully interoperating with components by appropriate interfaces (Chapter 5). The common feature of MR grid applications is that these applications are executed in two distinct phases, identified as *map* and *reduce*. All executions in the map phase are embarrassingly parallel. This means, the physically independent processors that execute separate parallel activities do not need to communicate

or synchronise. In contrast, the executions in the reduce phase involve some communication. Embarrassingly parallel grid applications are MR applications without the reduce phase. We refer to an activity that is executed during the map phase as a *map activity*, and the reduce phase as a *reduce activity*.

The difference among MR grid applications is that each application has its own, possibly unique, set of map and reduce activities. The degree of complexity of each map and reduce activity varies from one application to another. For example, in a Monte Carlo simulation, the map activity can be a random simulation that, depending on the data set or parameters, has highly variable long execution times. In other contexts, the map activity may simply be a linear scan through a given document whose execution time depends on the length of the document, e.g., a small file vs. a very large corpus of documents. Likewise, the reduce activity may be a simple addition in one case and a complex join in some very large global data structure in another.

The other important difference among MR grid applications is the way reduce activities are executed. Reduce activities could be executed sequentially, hierarchically, asynchronously, or randomly. The order of reduce activities execution has an impact on the performance of the overall computation. We will now discuss this issue in detail.

6.1.1.1 Parallelism

An idealized execution order of map and reduce activities is shown in Figure 6.1, where all map activities are executed first (level l), and then their outputs will successively be reduced by taking logarithmic steps (levels [0, l-1]). In practice, the execution of MR grid applications on a dynamic grid environment does not necessarily honour the static structure shown in Figure 6.1. Depending on the number of available resources, the type of messaging implementation (Hadoop [Had] vs. MPI [The Open MPI Development Team, 2011] vs. Xgrid [Xgr]), and/or the use of barriers between successive reduction steps, the order of execution activities could vary from one execution to another.

Map activities are embarrassingly parallel, therefore schedulers will find it easy to schedule these activities independently. Maximal parallelism among map activities is possible if the number of available cores, denoted by c, is at at least the same as the number of map activities, denoted by m. However, if $c < m$, assuming equal load distribution, each core is allocated m/c activities. Each core executes its allocated map activities sequentially, and then optionally reduces their outputs. We refer to the sequential execution of reduce activities by a core to

combine the outputs of its allocated map activities as a *local reduction*.

Reduce activities depend on map and previously completed reduce activities. Therefore, it is not possible to concurrently execute all reduce activities. These activities could be executed sequentially, hierarchically, or asynchronously. Local reduction is a sequential reduction. Once all cores complete their respective local reductions, their outputs will be further reduced one after another. We refer to such reduction as a *global sequential reduction*. The MPI_Reduce routine is an example of a global sequential reduction.

A *hierarchical reduction* is similar to what is depicted from level $l-1$ to level 0 in Figure 6.1. Once all of the cores complete the execution of map activities and local reductions, the outputs of the local reductions will be globally reduced in $log_2 c$ reduction steps. In each reduction step, maximal parallelism is possible. This is because executing the reduce activities in the first reduction step requires half of the cores that are used during the execution of the map activities; and the execution of reduce activities in each subsequent reduction step requires only half of the cores that are being used in the current reduction step.

The other form of parallelism among the activities of an MR application is achieved by interleaving the execution of map and reduce activities. If two cores, for instance, complete the execution of their allocated map activities and local reductions ahead other cores, then their output can be immediately reduced without having to wait for the other cores to finish their computations.

6.1.2 The MapReduce-specific RAC Approach

The *MapReduce-specific (MR-specific) RAC approach* is a RAC approach that provides customized fault tolerance support to grid applications whose communication and computation pattern falls under the MapReduce dwarf. MR-specific fault tolerance managers understand the role of map and reduce activities, and the dependency between the activities as defined by the type of parallelism that is used during the computation.

The MR-specific fault tolerance managers know that map activities are embarrassingly parallel, and the executions of these activities do not depend on the output of previously completed computations. Therefore, these managers do not need any other information other than the preferred fault tolerance strategy to handle the (impending) failure of map activities.

The MR-specific fault tolerance managers understand that the execution of a reduce activity depends on the output of previously completed map or reduce activities. These managers handle the (impending) failure of a reduce activity in two ways. First, if a reduce activity

fails, all activities on which the failed activity depends will be re-executed to get the input of the failed activity. If the re-execution is completed successfully, then the managers restart the failed activity. Second, the managers save the output of previously completed activities and meta-data that identifies the activity that produces a given output, on a persistent storage. If a reduce activity fails or is predicted to fail, then the MR-specific managers access the required data from the storage to execute either a reactive or a proactive strategy. Since the checkpointing strategy saves the current state of computation, if the proactive strategy is checkpointing, the managers do not need the output of previously completed activities. However, if the proactive strategy is replication, for example, executing a replica of an activity necessitates access to the input data that is needed for the computation.

Since the MR-specific fault tolerance managers identify the activities in an MR grid application as either map or reduce, the managers are able to use various kinds of fault tolerance strategies to handle the failure of each type of activity. For example, failure in map activities could be managed using restart while failure in reduce activities is managed by replication; the number of maximum retries to recover a failed map or reduce activity could be different; or the number of maximum retries to recover failed reduce activities could be increased as the overall computation gets closer and closer to completion in a hierarchical reduction.

6.1.3 MR-specific RACS

An *MR-specific RACS* is a fault tolerant grid system that realizes the MR-specific RAC approach. The reference architecture of an MR-specific RACS is similar to the one discussed in Section 5.1. The only difference between a RACS that is introduced in Section 5.1 and an MR-specific RACS is the head manager and compute managers of the MR-specific RACS are specifically designed to provide fault tolerance support to MapReduce grid applications. Since the behaviours of the head manager and compute managers are excluded from the formal models, the global behaviour of MR-specific RACS is also as explained in Section 5.2.

6.1.3.1 The MR-specific RACS Model

We modify `GSbsp`, the BSP-based RACS model shown in Figure 5.8, to reflect how an MR application execution would be carried out on a BSP computer. For such modification, we make the following simplifying assumptions:

- In the abstraction (and different from the concrete implementation), the execution of all map activities must be completed before any of the reduce activities begins. Therefore,

MapReduce application execution is represented by a sequence of two supersteps: map and reduce.

- Since all map activities are executed locally without requiring any communication, failure of a map activity is most likely caused by core or memory failure, not communication failure. Therefore, *in the map superstep, the global communication is assumed to be failure free.*

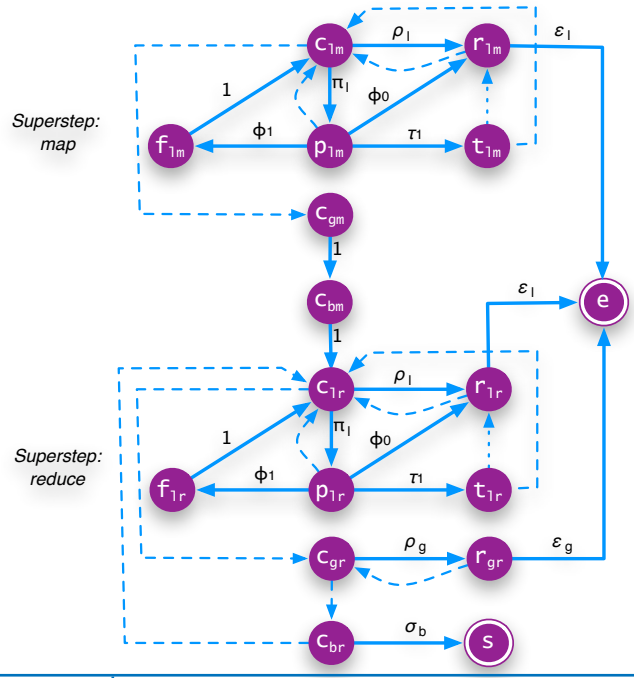
- Reduce activities are considered to be dominated by communication. Therefore, *failure in reduce activities is caused by core, memory failure and communication failure.*

- In both map and reduce supersteps, we assume the *barrier synchronisation to be failure free.*

The global behaviour of an MR-specific RACS is modelled by a parameterised DTMC. The model is a matrix of rank 17, and is denoted by `GMR`. Figure 6.2 shows the transition diagram of `GMR`. Except for states `e` and `s`, the states in the transition diagram are labelled as $\mathtt{x_i}$, where `x` is the corresponding state in `GSbsp` (Section 5.3.2), and `i` is an initial for the name of the superstep, i.e. `m` for a state in the map superstep, and `r` for a state in the reduce superstep. For example, $\mathtt{c_{lm}}$ is a compute state that represents local computations in the map superstep. The parameters of `GMR` are $\rho_{_l}$, $\rho_{_g}$, $\epsilon_{_l}$, $\epsilon_{_g}$, $\pi_{_l}$, $\tau_{_l}$, $\phi_{_l}$, $\phi_{_0}$, α, and θ.

In the map superstep, `GMR` transits from $\mathtt{c_{lm}}$ to $\mathtt{c_{gm}}$ if all map activities are successfully completed. A failure will occur during the execution of map activities with probability of $\rho_{_l}$. This makes `GMR` transit from $\mathtt{c_{lm}}$ to $\mathtt{r_{lm}}$. The failed map activities will, with probability of $\epsilon_{_l}$, never recover. During the execution of map activities, a prediction is made with probability of $\pi_{_l}$— $\mathtt{GMR}(\mathtt{c_{lm}}, \mathtt{p_{lm}})$. The prediction will be positive with probability of $\tau_{_l} + \phi_{_l}$. In such cases, `GMR` transits from $\mathtt{p_{lm}}$ to either $\mathtt{t_{lm}}$ or $\mathtt{f_{lm}}$, depending on the correctness of the prediction. For negative predictions, a true negative prediction causes the transition of `GMR` from $\mathtt{p_{lm}}$ to $\mathtt{c_{lm}}$ while a false negative prediction causes the transition of `GMR` from $\mathtt{p_{lm}}$ to $\mathtt{r_{lm}}$ (see Section 5.2.3 for a detailed discussion about the behaviour of a RACS after a prediction is made).

In line with assumptions stated in Section 6.1.3, after the successful execution of all map activities—$\mathtt{GMR}(\mathtt{c_{lm}}, \mathtt{c_{gm}})$, `GMR` transits to $\mathtt{c_{lr}}$ with probability of 1. All the states and the transitions in the map superstep are also present in the reduce superstep. However, the reduce superstep has an additional state, $\mathtt{r_{gr}}$, along with the state's outgoing and incoming transitions, and one more outgoing transition from the barrier synchronisation state, $\mathtt{GMR}(\mathtt{c_{br}}, \mathtt{c_{lr}})$.



(i) → (j)	Description
———	GMR(i,j) is a free variable
— —	GMR$(i,j) = 1 - \sum_{k(k\neq j)}$ GMR(i,k)
·····	GMR(i,j) is an expression in free variables

Figure 6.2: The parameterised DTMC of an MR-specific RACS (GMR)

- Since the execution of reduce activities involves a lot of communication, there is ρ_g probability for a failure to occur during communication, and cause the transition of GMR from c_{gr} to r_{gr}. The failed communication can never be re-established with probability of ϵ_g.

- Depending on the type of parallelism, the reduce superstep could be iterated multiple times—GMR(c_{br}, c_{lr}). For example, in hierarchical reduction, GMR transits from c_{br} to c_{lr} as many times as $log_2 c - 1$, where c is the number of cores that executed the map activities, provided that the overall computation completes successfully.

6.1.3.2 Reliability Prediction

We use GMR to predict the reliability of a MapReduce grid application to which an MR-specific RACS provides fault tolerance support. The properties of GMR are given below:

- $\mathtt{c_{lm}}$, $\mathtt{r_{lm}}$, $\mathtt{p_{lm}}$, $\mathtt{f_{lm}}$, $\mathtt{t_{lm}}$, $\mathtt{c_{gm}}$, $\mathtt{c_{bm}}$, $\mathtt{c_{lr}}$, $\mathtt{r_{lr}}$, $\mathtt{p_{lr}}$, $\mathtt{f_{lr}}$, $\mathtt{t_{lr}}$, $\mathtt{c_{gr}}$, $\mathtt{r_{gr}}$ and $\mathtt{c_{br}}$ are transient states.
- $\mathtt{c_{lm}}$ is the entry state and $\mathtt{c_{br}}$ is the exit state.
- s and e are absorbing states.
- The reliability of the MapReduce grid application is the transition probability of GMR from $\mathtt{c_{lm}}$ to s in the underlying absorption probability matrix.

We used MATLAB [MathWorks Website] to symbolically derive the absorption probability matrix of GMR. Due to the size of the reliability equation, we do not show the equation here.

6.1.3.3 Overhead

We use GMR to estimate the overhead of the fault tolerance management of an MR-specific RACS. In the MR-specific RACS, the overhead comes from executing a reactive strategy, a proactive strategy and making predictions. Therefore, the overhead of an MR-specific RACS is the weighted sum of the expected number of visits from $\mathtt{c_{lm}}$ to $\mathtt{r_{lm}}$, $\mathtt{r_{lr}}$, $\mathtt{r_{gr}}$, $\mathtt{p_{lm}}$, $\mathtt{p_{lr}}$, $\mathtt{f_{lm}}$, $\mathtt{f_{lr}}$, $\mathtt{t_{lm}}$ and $\mathtt{t_{lr}}$. The total number of visits to each of these states is weighted by the overhead of a single visit to each state. The expected number of visits are obtained from the potential matrix of GMR.

Let PMR be the potential matrix of GMR, and $O.\mathtt{i}$ be the overhead of a single visit to state i. Equation (6.1) shows the overhead of a RACS, denoted by OMR.

$$\begin{aligned}\mathtt{OMR} = {} & \mathtt{PMR}(\mathtt{c_{lm}}, \mathtt{r_{lm}}) \times O.\mathtt{r_{lm}} + \mathtt{PMR}(\mathtt{c_{lm}}, \mathtt{r_{lr}}) \times O.\mathtt{r_{lr}} + \mathtt{PMR}(\mathtt{c_{lm}}, \mathtt{r_{gr}}) \times O.\mathtt{r_{gr}} \\ & + \mathtt{PMR}(\mathtt{c_{lm}}, \mathtt{p_{lm}}) \times O.\mathtt{p_{lm}} + \mathtt{PMR}(\mathtt{c_{lm}}, \mathtt{p_{lr}}) \times O.\mathtt{p_{lr}} + \mathtt{PMR}(\mathtt{c_{lm}}, \mathtt{f_{lm}}) \times O.\mathtt{f_{lm}} \\ & + \mathtt{PMR}(\mathtt{c_{lm}}, \mathtt{f_{lr}}) \times O.\mathtt{f_{lr}} + \mathtt{PMR}(\mathtt{c_{lm}}, \mathtt{t_{lm}}) \times O.\mathtt{t_{lm}} + \mathtt{PMR}(\mathtt{c_{lm}}, \mathtt{t_{lr}}) \times O.\mathtt{t_{lr}}\end{aligned} \tag{6.1}$$

6.1.4 Evaluating the MR-specific RAC approach

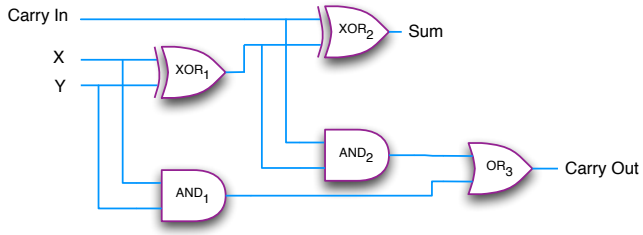
We evaluate the reliability-overhead tradeoff that is enabled by the MR-specific RAC approach to MapReduce grid applications in Chapter 9. The empirical evaluation of the approach is presented in Section 9.1.1.

6.2 Combinational Logic Dwarf

Asanovic et al. [2006] defined the Combinational Logic (CL) dwarf as dataflow networks of functions, where functions are logical and have stored state, more specifically: "*Combinational Logic generally involves performing simple operations on very large amounts of data often exploiting bit-level parallelism.*"

In fact, CL is a well known pattern for digital circuits involving interconnected boolean functions that process streams of binary data [Wakerly, 2000]. The outputs of these functions depend only on the current inputs. The dataflow in such computation is represented by logic gates, a mathematical expression, a truth table, and/or a schematic diagram.

Figure 6.3 shows bit level addition of two binary numbers, denoted by X and Y, using the `Full Adder` circuit. For each bit-pair addition, the initial inputs of the circuit are x_i, y_i, and a carry in, denoted by z_i. The final outputs are the sum of x_i, y_i and z_i, and a carry out. `Full Adder` repeats this computation until the last bit-pair of X and Y is added.



(a) Logic Gates

Input			Output	
X	Y	Carry In	Carry Out	Sum
0	0	0	0	0
1	1	0	1	0
1	0	1	1	0
0	0	1	0	1
⋮	⋮	⋮	⋮	⋮

(b) Truth Table

Figure 6.3: `Full Adder`: CL example in hardware

The functions in `Full Adder` can be executed in parallel using the principle of Multiple-Instruction-Single-Data of Flynn's taxonomy [Flynn, 1972]. Whenever a new bit-pair of X and Y arrives, $\texttt{XOR}_1$ and $\texttt{AND}_1$ are executed in parallel. Once the computations in $\texttt{XOR}_1$ and $\texttt{AND}_1$ are completed, the outcome of the computations is fed to the next functions; the output of $\texttt{XOR}_1$ is an input to $\texttt{XOR}_2$ and $\texttt{AND}_2$, and the output of $\texttt{AND}_1$ is an input to $\texttt{OR}_3$. Then, $\texttt{XOR}_2$ and $\texttt{AND}_2$ will be executed in parallel to produce the sum of the bit-pair, and the input to the last function, respectively. Finally, $\texttt{OR}_3$ is executed to produce the carry out.

6.2.1 Combinational Logic Grid Applications

The principles of CL at a hardware parallelism level can be exploited for higher-level software coordination in grid applications through the CL dwarf. *Combinational Logic grid applications* are, therefore, grid programs that are composed of dataflow networks of functions that operate on streams of very large amounts of data. In CL hardware, binary bits are streamed, whereas in CL grid applications, streams include bits, numerical figures, satellite images and other rich datasets. The functions in CL hardware are simple binary operations, while functions in a CL grid application could be simple like `XOR` or complex like rendering digital graphics.

CL applications are available in embedded computing, machine learning and databases [Asanovic et al., 2006]. Examples for CL applications abound, [NIST, 2001, 1999, Peterson and Brown, 1961, Garotte and Bras, 1995, Wang and Rundensteiner, 2009, Kuntschke et al., 2006]. Computing Cyclic Redundancy Codes (CRC) and RSA encryption for data integrity and security. High-performance computing of satellite/radio signal streams for weather forecasting, environmental modelling or air/border control exhibits the CL pattern. Data streams from large widely distributed sensor networks, such as RFID, in transport and logistics, or in content-based routing systems are another example where increasingly vast amounts of real-time data require parallel processing.

We view CL grid applications as directed acyclic graphs (DAGs) of processing steps. A *processing step* represents the concurrent execution of mutually exclusive functions. The partial order of the DAG represents the control and dataflow dependencies, i.e. both synchronisation and communication. Each vertex in the DAG represents a function, hereafter referred to as a *CL activity*. CL activities are connected by streams. The minimal elements of the DAG are fed by *input streams* and the maximal elements generate the *output streams* of the dataflow network. The dataflow network represents the highest level of coordination abstraction in the CL pattern.

Figure 6.4 shows the DAG of a CL grid application. The CL grid application, denoted by `E`, is our running example. `E` has three processing steps, two input streams, $\{I_0, I_1\}$, two output streams, $\{O_5, O_6\}$, and seven CL activities, $\{a_0, a_1, a_2, a_3, a_4, a_5, a_6\}$. For each element of the input streams, computation begins in processing step 1, and ends in processing step 3. When data arrives via I_0 and I_1, the CL activities in processing step 1, a_0 and a_1, begin execution. The outputs of a_0 and a_1 are inputs to activities in processing step 2, i.e. a_2, a_3 and a_4. CL activity a_2 depends on both a_0 and a_1, while a_3 and a_4 depend only on a_1. Upon the completion of a_2, a_3 and a_4, the outputs of the activities is used to commence the execution

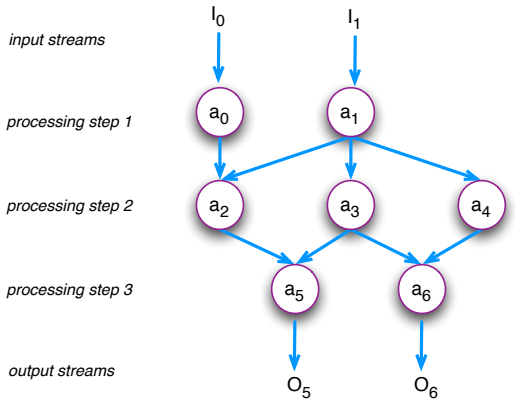


Figure 6.4: The DAG of a CL grid application `E`: the minimal elements a_0 and a_1 form the first processing step, followed by $\{a_2, a_3, a_4\}$, and finally $\{a_5, a_6\}$. The input streams to the DAG are shown at the top of the figure, the outputs at the bottom. The dependencies (direct partial ordering) between processing steps are implicitly associated with input and output streams. For example the dependency $a_1 \rightarrow a_3$ is associated with O_1 and I_3.

of a_5 and a_6 in processing step 3. CL activity a_5 depends on a_2 and a_3, while a_6 depends on a_3 and a_4. The computation ends when a_5 and a_6 produce O_5 and O_6, respectively.

6.2.1.1 Parallelism

The execution of a CL grid application involves one or more of the following parallelism types: *Course-Grain*, *Pipeline*, *Repeated* and *Stream*. Though two or more parallelism types can be used during CL application execution, our discussion is focused on the use of a single type of parallelism at a time.

6.2.1.1.1 Coarse-Grain Parallelism

Course-Grain parallelism is a type of parallelism that facilitates the concurrent execution of mutually exclusive activities. Such parallelism type is used to execute CL activities in a single processing step. Table 6.1 shows the execution of `E` using coarse-grain parallelism. When the n^{th} data element of I_0 and I_1 arrives, activities in processing step 1 are concurrently executed to process the data. This is followed by the concurrent execution of activities in processing step 2, and then the concurrent execution of activities in processing step 3. Once the output is produced, the computation is repeated for the $n + 1^{st}$ data element, and for $n + 2^{nd}$ data element, and so on.

Table 6.1: The execution of `E` using coarse-grain parallelism. Activities in each row are executed in parallel.

Input stream element	Processing step	CL Activities	CL Superstep
n^{th}	1	a_{0_n}, a_{1_n}	1
n^{th}	2	a_{2_n}, a_{3_n}, a_{4_n}	2
n^{th}	3	a_{5_n}, a_{6_n}	3
$n+1^{st}$	1	$a_{0_{n+1}}$, $a_{1_{n+1}}$	4
$n+1^{st}$	2	$a_{2_{n+1}}$, $a_{3_{n+1}}$, $a_{4_{n+1}}$	5
$n+1^{st}$	3	$a_{5_{n+1}}$, $a_{6_{n+1}}$	6
$\vdots$	$\vdots$	$\vdots$	$\vdots$

6.2.1.1.2 Pipeline Parallelism

Pipeline parallelism is a type of parallelism that executes all processing steps of a CL grid application concurrently, provided that the activities in each processing step work on different elements of the input stream. This type of parallelism is possible due to the streaming nature of the CL dataflow network. While CL activities in processing step i are working on the n^{th} element of the input stream, CL activities in processing step $i+1$ can already start working on the $n-1^{st}$ element of the input stream, CL activities in processing step $i+2$ can already start working on the $n-2^{nd}$ element of the input stream, and so on. Table 6.2 shows the execution of `E` using pipeline parallelism.

Table 6.2: The execution of `E` using pipeline parallelism. Activities in each row are executed in parallel.

Input stream element	Processing step 1		Processing step 2			Processing step 3		CL Superstep
n^{th}	a_{0_n}	a_{1_n}	$a_{2_{n-1}}$	$a_{3_{n-1}}$	$a_{4_{n-1}}$	$a_{5_{n-2}}$	$a_{6_{n-2}}$	1
$n+1^{st}$	$a_{0_{n+1}}$	$a_{1_{n+1}}$	a_{2_n}	a_{3_n}	a_{4_n}	$a_{5_{n-1}}$	$a_{6_{n-1}}$	2
$n+2^{nd}$	$a_{0_{n+2}}$	$a_{1_{n+2}}$	$a_{2_{n+1}}$	$a_{3_{n+1}}$	$a_{4_{n+1}}$	a_{5_n}	a_{6_n}	3
$n+3^{rd}$	$a_{0_{n+3}}$	$a_{1_{n+3}}$	$a_{2_{n+2}}$	$a_{3_{n+2}}$	$a_{4_{n+2}}$	$a_{5_{n+1}}$	$a_{6_{n+1}}$	4
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

6.2.1.1.3 Repeated Parallelism

Repeated parallelism is a type of parallelism that simultaneously processes k sets of input streams using k identical CL applications. Many practical CL computations exhibit multiple repeated occurrences of the same CL processing pattern. For example in transport, image streams of toll gates come from more than one camera. Thus, the streaming of images from different sections of a freeway will occur in parallel. The subsequent processing of these multiple image streams can utilise the logical parallelism by *repeated* parallel processing steps. Therefore, an independent CL computation can be scheduled to process each image stream. Table 6.3 shows the execution of `E` using repeated parallelism when there are two sets of input streams. Since there are two sets of input streams, the activities in each processing step will have duplicates. For example, in processing step 1, the activities whose input comes from r are a_{0r} and a_{1r}, while the ones whose input is provided by p are a_{0p} and a_{1p}. Under the assumption that data size and arrival rate in all k sets of input streams are roughly the same, the execution of all k CL applications be in the same processing step at any given time. For example, when the n^{th} element arrives from r and p, then a_{0r_n}, a_{1r_n}, a_{0p_n} and a_{1p_n} are concurrently executed.

Table 6.3: The execution of `E` using repeated parallelism with 2 sets of input streams. Activities in each row are executed in parallel.

Input stream element	**Processing step**	**Input stream set r**	**Input stream set p**	**CL Superstep**
n^{th}	1	a_{0r_n}, a_{1r_n}	a_{0p_n}, a_{1p_n}	1
n^{th}	2	a_{2r_n}, a_{3r_n}, a_{4r_n}	a_{2p_n}, $a_{3k_{2n}}$, a_{4p_n}	2
n^{th}	3	a_{5r_n}, a_{6r_n}	a_{5p_n}, a_{6p_n}	3
$n+1^{st}$	1	$a_{0r_{n+1}}$, $a_{1r_{n+1}}$	$a_{0p_{n+1}}$, $a_{1p_{n+1}}$	4
$n+1^{st}$	2	$a_{2r_{n+1}}$, $a_{3r_{n+1}}$, $a_{4r_{n+1}}$	$a_{2p_{n+1}}$, $a_{3p_{n+1}}$, $a_{4p_{n+1}}$	5
$n+1^{st}$	3	$a_{5r_{n+1}}$, $a_{6r_{n+1}}$	$a_{5p_{n+1}}$, $a_{6p_{n+1}}$	6
⋮	⋮	⋮	⋮	⋮

6.2.1.1.4 Stream Parallelism

Stream parallelism is a type of parallelism that divides a single set of input stream into m slices and executes each slice using m identical CL applications. For stateful CL activities, there are generally *bounds on their history sensitivity*. This means, one can divide an input stream into

successive slices of the same size, such that any two slices can be reordered, while the order of elements in a slice is significant. In practice, this is done by windows or markers. For example, in some pattern recognition algorithms the size of a pattern match may be bounded and an input splitter keeps feeding fixed-length substrings from the input stream to different buffers used by parallel pattern matchers, starting from successive position j, $j+1$, $j+2$, etc. In other algorithms, markers define boundaries where such reordering can occur due to the independence of the data.

We *note* the similarity and the difference between stream and repeated parallelism types. Both exploit the independence of data to provide logical parallelism. The difference between the two arises in the way the data independence is achieved. In stream parallelism, the data independence is achieved by dividing a *single* stream into chunks of data elements. Each chunk of data is processed simultaneously. In repeated parallelism, data is streamed from *multiple* sources. The data from each stream source is computed simultaneously. Since, other than the way how the data is feed to the system, the style of the computation in both parallelism types is similar, stream parallelism will not be discussed further.

6.2.1.2 CL supersteps

The type of parallelism, along with the dependency structure (DAG), determines the group of CL activities that should be executed in parallel. For instance, activities of a *single* processing step are concurrently executed using coarse-grain parallelism, whereas activities of *all* processing steps run in parallel using pipeline parallelism. Inspired by the BSP model (see Section 4.2), we refer to the concurrent execution of a group of mutually exclusive CL activities as a *CL superstep*. See the CL supersteps of `E` in Tables 6.1-6.3.

Under the following simplifying assumptions, without loss of generality, we view a CL computation as a sequence of CL supersteps:

- CL activities are relatively uniform in execution time.
- Alternatively, where there is large variation in execution times, the number of activities in a CL superstep is considerably larger than the number of available cores. This allows grid schedulers to randomly distribute CL activities to cores giving statistically uniform global behaviour in execution time.
- Data types are uniform and data sizes are roughly equal. This allows to make reasonably accurate characterization of the cost of collective communication that involves global

data transfers, and local peer-to-peer communication.

- If more than one set of input streams are used, which is the case in repeated parallelism, data arrival rate from all sets of input streams is roughly the same.

6.2.2 The Combinational Logic-specific RAC Approach

The *Combinational Logic-specific (CL-specific) RAC approach* is a RAC approach that provides customized fault tolerance support to grid applications whose communication and computation pattern falls under the Combinational Logic dwarf. CL-specific fault tolerance managers understand that a CL computation is a sequence of CL supersteps, and thus exploit superstep barriers to provide improved reliability without a heavy penalty on the cost of the overall computation. Unlike MapReduce grid applications, whose activities are identified either as `map` or `reduce` components, CL grid applications do not have a fixed number of component types. Therefore, CL-specific managers do not provide component type based fault tolerance support.

CL-specific managers use superstep barriers to save the outputs of CL activities in the current superstep on a persistent storage. The saved data is used to recover a failed CL activity. Suppose `E` is executed using coarse-grain parallelism. When the activities in CL superstep 1 complete working on the n^{th} data element, the outputs of a_{0_n} and a_{1_n} are written in a disk. Then, during the execution of activities in CL superstep 2, suppose a_{2_n} fails, and no proactive strategy is executed to avert the failure. In order to recover a_{2_n}, the CL-specific manager needs to identify the activities on which a_{2_n} depends. This is achieved using the DAG of `E`. Once the manager identifies the activities on which a_{2_n} depends, i.e., a_{0_n} and a_{1_n}, the manager retrieves their outputs from the storage and restarts a_{2_n}. It is important to note that if the outputs of completed CL activities on which a failed CL activity depends are not saved, CL-specific managers will re-execute the failed activity as well as the activities on which the failed activity depends.

CL managers also use superstep barriers to remove any information that is no longer needed. The objective of discarding unwanted data is to prevent any potential storage problem. Since a CL grid application execution involves large streams of data, it is a matter of time before the available disk fills up if all the data that are saved are left untouched. Suppose `E` is executed using coarse-grain parallelism. When the activities in CL superstep 2 complete working on the n^{th} data element, the CL managers save the outputs of a_{2_n}, a_{3_n} and a_{4_n} and discard any information about a_{0_n} and a_{1_n}. An alternative approach to this is to delete all information that is associated with the computation of an element of an input stream when the compu-

tation of the element reaches its final stage. As shown in Table 6.1, the processing of the n^{th} element of the input stream reaches its final stage in CL superstep 3. Therefore, when the activities in CL superstep 3 complete execution, all data that is saved in relation to CL activities $a_{0_n}, a_{1_n}, a_{2_n}, a_{3_n}$ and a_{4_n} will be removed.

If a CL application is executed with coarse-grain parallelism, then CL-specific managers need to keep track of computational information about one data element. In pipeline parallelism, for a CL application with d processing steps, the managers save information about d data elements at a time. In the case of repeated parallelism, the amount of information to be saved depends on the number of the set of input streams. The CL-specific managers keep such information until the processing of the data elements is either successfully completed or irrecoverably failed.

6.2.3 CL-specific RACS

A *CL-specific RACS* is a fault tolerant grid system that realises the CL-specific RAC approach. The reference architecture of a CL-specific RACS is similar to the one discussed in Section 5.1. The only difference between a RACS that is introduced in Section 5.1 and a CL-specific RACS is the head manager and compute managers of the CL-specific RACS are specifically designed to provide fault tolerance support to CL grid applications. Since the behaviours of the head manager and compute managers are excluded from the formal models, the global behaviour of CL-specific RACS is also as explained in Section 5.2.

6.2.3.1 The CL-specific RACS Model

We unfold `GSbsp`, the BSP-based RACS model shown in Figure 5.8, to represent the execution of a CL grid application as a sequence of CL supersteps on a BSP computer. In the model, each CL superstep has

- *local computations*, which represent the execution of CL activities of the CL superstep,
- *global communications*, which represent the transfer of outputs of from the CL activities in the current superstep to the CL activities in the next CL superstep, and
- a *barrier synchronisation*, which ensures that no computation in the next superstep commences before the completion of *all* local computations and global communications in the current superstep.

In order to unfold GSbsp, the total number of the CL supersteps during the execution of a CL grid application should be known. Unfortunately, due to the involvement of data streams in CL computations, there are infinite CL supersteps. For E, this is shown in Tables 6.1-6.3. Nonetheless, we observe that, starting from arbitrary CL superstep n, the type of activities in all CL supersteps that are identified by $d \times n$, for some natural number d, are the same. The difference between these supersteps is that the activities in each superstep will be working on different elements of the input stream. In coarse-grain and repeated parallelism, d is the depth of the DAG of the CL application, while in pipeline parallelism, $d = 1$. For example, if the execution of E involves either coarse-grain or repeated parallelism, starting from an arbitrary CL superstep, every third superstep will contain a similar type of CL activities. This is because the depth of the DAG of E is three. As shown in Table 6.1, supersteps 1 and 4 contain a_0 and a_1, where the activities in the respective supersteps work on the n^{th} and $n + 1^{st}$ elements of the input stream. If E is executed using pipeline parallelism, all CL activities are executed in every CL superstep. This is shown in Table 6.2.

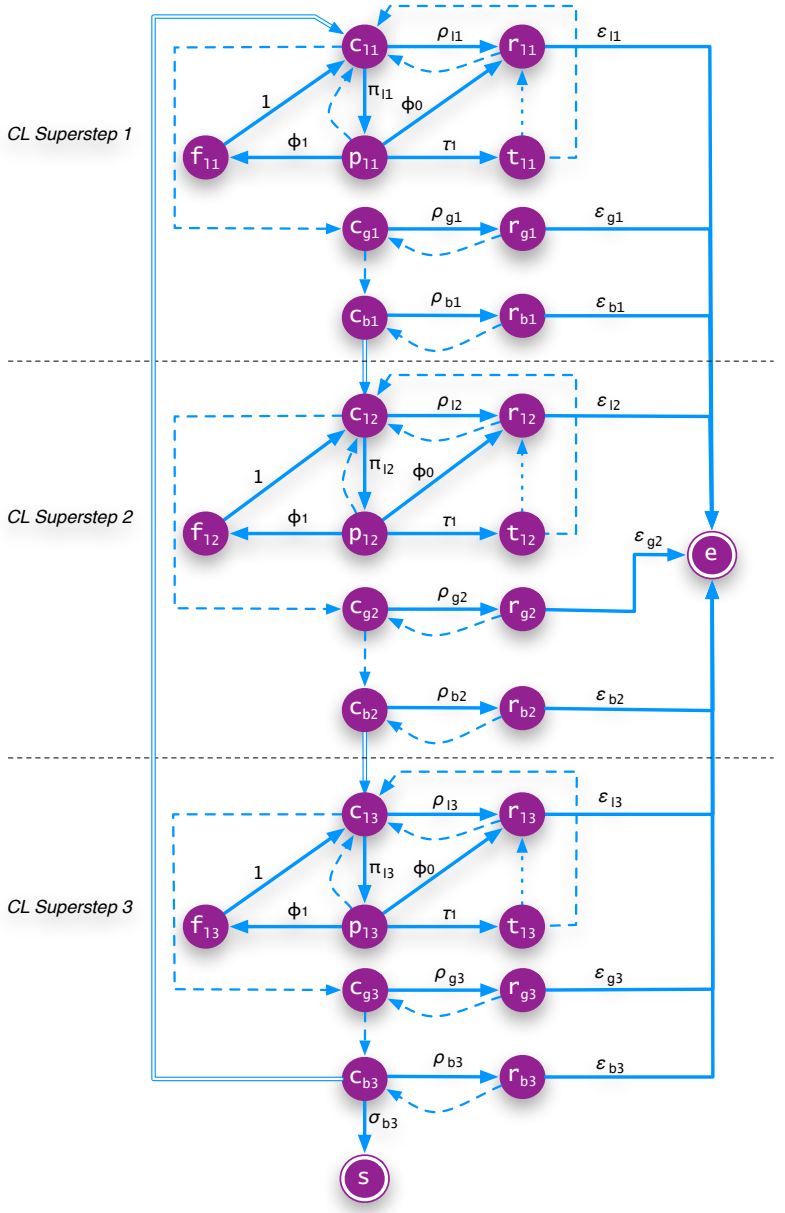
Despite the execution of a CL application having infinite number of CL supersteps, since similar computation pattern is repeated, it is possible to unfold GSbsp using finite number of CL supersteps as long as the depth of the DAG of the application and/or the type of parallelism is known. For example, since we have established that executing E with either coarse-grain or repeated parallelism is equivalent to executing three unique CL supersteps repeatedly, with different data element each time, the refined GSbsp that represents such computation will consist of three CL supersteps. The transition diagram of the refined model is shown in Figure 6.5. On the other hand, if E is executed with pipeline parallelism, since there is only one unique CL superstep, no further refinement of GSbsp is needed, as GSbsp already represents a grid computation with a single superstep.

The transition diagram of the refined GSbsp for any CL application execution, denoted by GCL, is shown in Figure 6.6.

6.2.3.2 Reliability Prediction

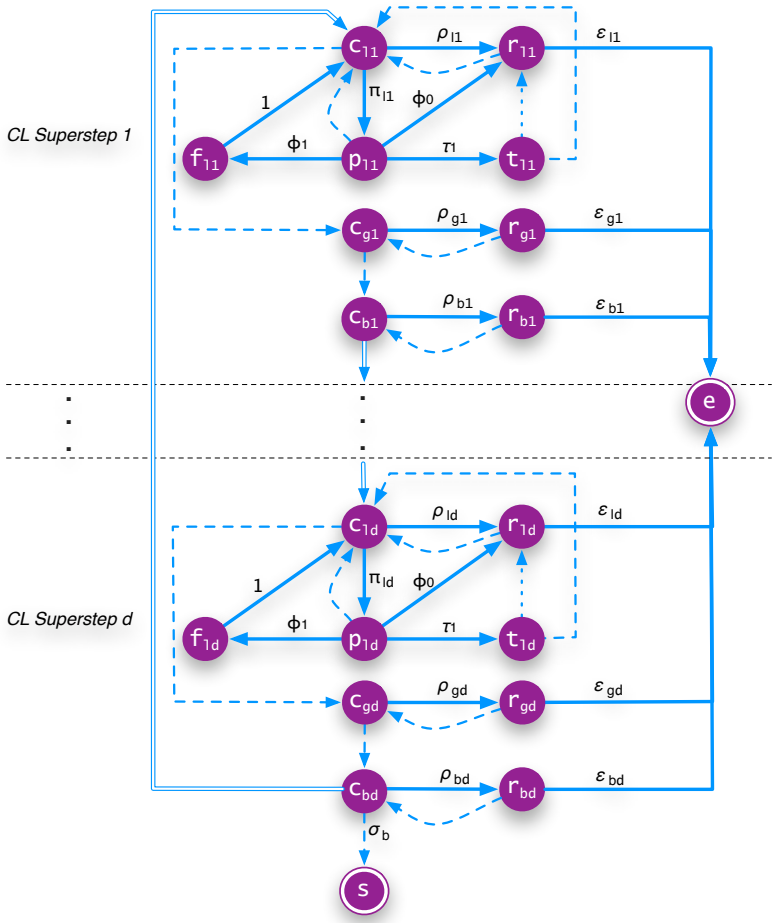
We use GCL to predict the reliability of a RAC-based CL grid application to which a CL-specific RACS provides fault tolerance support. The properties of GCL are given below:

- $c_{l1}, c_{l2}, \ldots, c_{ld}$; $r_{l1}, r_{l2}, \ldots, r_{ld}$; $p_{l1}, p_{l2}, \ldots, p_{ld}$, $f_{l1}, f_{l2}, \ldots, f_{ld}$; $t_{l1}, t_{l2}, \ldots, t_{ld}$; $c_{g1}, c_{g2} \ldots, c_{gd}$; $r_{g1}, r_{g2}, \ldots, r_{gd}$; $c_{b1}, c_{b2}, \ldots, c_{bd}$; $r_{b1}, r_{b2}, \ldots, r_{bd}$ are transient states, where d is total number of unique CL supersteps.



(i) → (j)	Description
=	a transition to the next superstep

Figure 6.5: The transition diagram of the refined `GSbsp` for `E` with either coarse-grain or repeated parallelism.



(i) → (j)	Description
———	$\texttt{GCL}(i,j)$ is a free variable
– – –	$\texttt{GCL}(i,j) = 1 - \sum_{k(k \neq j)} \texttt{GCL}(i,k)$
. . . . .	$\texttt{GCL}(i,j)$ is an expression in free variables
=	a transition to the next superstep

Figure 6.6: The parameterised DTMC of a CL-specific RACS (GCL)

- $\texttt{c}_{11}$ is the entry state and $\texttt{c}_{\texttt{bd}}$ is the exit state.
- s and e are absorbing states.
- The reliability of the grid application is the transition probability of GCL from $\texttt{c}_{11}$ to s in the underlying absorption probability matrix.

We used MATLAB [MathWorks Website] to symbolically derive the absorption probability matrix of GCL. Due to the size of the reliability equation, we do not show the equation here.

6.2.3.3 Overhead

We use GCL to estimate the overhead of the fault tolerance management of a CL-specific RACS. In the RACS, the overhead comes from executing a reactive strategy, a proactive strategy and making predictions. Therefore, the overhead of a CL-specific RACS is the weighted sum of the expected number of visits from $\mathtt{c_{l1}}$ to $\mathtt{r_{lj}}$, $\mathtt{r_{gj}}$, $\mathtt{r_{bj}}$, $\mathtt{f_{lj}}$, $\mathtt{t_{lj}}$ and $\mathtt{p_{lj}}$, for all superstep j. The total number of visits to each of these states is weighted by the overhead of a single visit to each state. The expected number of visits are obtained from the potential matrix of GCL.

Let PCL be the potential matrix of GCL, and $O.\mathtt{i}$ be the overhead of a single visit to state i. Equation (6.2) shows the overhead of a CL-specific RACS, denoted by OCL.

$$\begin{aligned}\mathtt{OCL} = &\sum_{j=1}^{d}(\mathtt{PCL}(\mathtt{c_{l1}},\mathtt{r_{lj}}) \times O.\mathtt{r_{lj}} + \mathtt{PCL}(\mathtt{c_{l1}},\mathtt{r_{gj}}) \times O.\mathtt{r_{gj}} + \mathtt{PCL}(\mathtt{c_{l1}},\mathtt{r_{bj}}) \times O.\mathtt{r_{bj}}) \\ &+ \sum_{j=1}^{d}(\mathtt{PCL}(\mathtt{c_{l1}},\mathtt{f_{lj}}) \times O.\mathtt{f_{lj}} + \mathtt{PCL}(\mathtt{c_{l1}},\mathtt{t_{lj}}) \times O.\mathtt{t_{lj}} + \mathtt{PCL}(\mathtt{c_{l1}},\mathtt{p_{lj}}) \times O.\mathtt{p_{lj}})\end{aligned} \tag{6.2}$$

where d is total number of unique CL supersteps

6.2.4 Evaluating the CL-specific RAC approach

We evaluate the reliability-overhead tradeoff that is enabled by the CL-specific RAC approach to Combinational Logic grid applications in Chapter 9. The empirical evaluation of the approach is presented in Section 9.1.2.

6.3 Summary

In this chapter, the architecture-specific RAC approach was introduced. We also presented the generic RAC approach and its limitations. We precisely characterised MR and CL grid applications. Along with the characterisation of MR applications, we discussed local reductions, global sequential reductions, hierarchical reduction and interleaving parallelism. Likewise, in CL, we discussed coarse-grain, pipeline, repeated and stream parallelism. We also discussed how to manipulate the architecture of MR and CL application to provide better fault tolerance support via the MR-specific and CL-specific RAC approaches, respectively. Further, we showed

that, along with reliability prediction and overhead equations, the adaptation of the BSP-based RACS model to accurately represent the MR-specific RACS and the CL-specific RACS.

Part III

Evaluation

Friend to Groucho Marx: "Life is difficult!"
Marx to Friend: "Compared to what?"

In Part II, the generic and the architecture-specific RAC approaches were introduced. We studied how to provide customized fault tolerance support to MapReduce (MR) and Combinational Logic (CL) grid applications in-depth. We now evaluate the RAC approach with respect to reliability and overhead. We first describe our experiment testbed in Chapter 7, and then present our experiment design in Chapter 8. In Chapter 9, we report the reliability-overhead tradeoffs that are enabled by the generic, the MR-specific and the CL-specific RAC architectures, and show how these tradeoffs are affected by the choice of a fault tolerance strategy, the parameters of the selected fault tolerance strategy, and the prediction interval and accuracy. Our evaluation is based on simulations and real runs.

Chapter 7

Experiment Testbed

We designed an experiment testbed to answer our research questions (Section 1.1). The testbed provides a platform to evaluate the reliability-overhead tradeoff of the generic, the MR-specific and the CL-specific RAC architectures, and to analyse the sensitivity of such tradeoffs to prediction accuracy, the choice of a fault tolerance strategy, and the parameters of the selected fault tolerance strategy. In the testbed, each RAC architecture can be paired with restart, checkpointing-rollback or replication fault tolerance strategy.

The testbed executes a grid application in either virtual-time or real-time. During such execution, the testbed provides simulated fault tolerance support, which is based on the principles of the RAC approach, to the grid application. When the execution of the grid application is completed, the testbed outputs simulated reliability, simulated *execution time*, which is the real time that would have been elapsed from the start to end of the computation, and simulated *overall cost*, which is the total CPU time that would have been consumed by the computation. Suppose a computation has two embarrassingly parallel activities, which each runs for three seconds. Assuming the two activities are executed concurrently, the total *execution time* of the computation is three seconds, while the *cost* of the computation is six seconds.

The experiment testbed is parameterised in order to allow the user to explore a representative set of scenarios reflecting the space of all possible grid fault tolerance scenarios. Not all input parameters of the testbed are used in every experiment run. The use of a parameter depends on the type of the grid application to which fault tolerance support is provided (MapReduce or Combinational Logic), and the required type of fault tolerance support (prediction-based, reactive-only, proactive-only, and/or hybrid).

Table 7.1: Parameter table

Type	Name	Values	Experiments	Variables
Basic	Type of execution	virtual-time, real-time	Shared parameters in all experiments	Controlled*
	Failure seed	$\mathbb{Z}$		
	Recovery seed	$\mathbb{Z}$		
	Experiment size	$\mathbb{Z}^+$		
Grid Infrastructure	No. nodes	$\mathbb{Z}^+$		
	No. cores per node	$\mathbb{Z}^+$		
Grid Application	Global communication overhead	$\mathbb{Q}^+$		
	Barrier synchronisation overhead	$\mathbb{Q}^+$		
	Probability of activity failure	$[0, 1]$		
	Type of global reduction	sequential, hierarchical	Parameters used only in MR experiments	
	No. map activities	$\mathbb{Z}^+$		
	Map activity execution time	$\mathbb{Q}^+$		
	Reduce activity execution time	$\mathbb{Q}^+$		
	CL activity execution time	$\mathbb{Q}^+$	Parameters used only in CL experiments	
	Dependency structure	text file		
	Type of parallelism	pipeline, course-grain, repeated		
	No. sets of input streams	$\mathbb{Z}^+$		
	No. stream elements	$\mathbb{Z}^+$		
Fault Tolerance Management	No. maximum retry	$\mathbb{Z}^+$	Shared parameters in all experiments	Independent†
	Type of RAC architecture	generic, MR-specific, CL-specific		
	Type of fault tolerance strategy	restart, replication, checkpointing-rollback		
	Probability of unrecoverable failure	$[0, 1]$		
	Probability of false negatives	$[0, 1]$	Prediction Based	
	Probability of false positives	$[0, 1]$		
	Prediction interval	$\mathbb{Q}^+$		
	Prediction overhead	$\mathbb{Q}^+$		
	Replica overhead	$\mathbb{Q}^+$	Replication	
	Checkpointing cost	$\mathbb{Q}^+$	Checkpointing	
	Rollback cost	$\mathbb{Q}^+$		
	Restart overhead	$\mathbb{Q}^+$	Restart	

* A variable whose value is held constant in all equivalent experiments.

† A variable whose value is varied in equivalent experiments.

7.1 Parameters

The parameters of the experiment testbed are broadly divided into four types: *basic*, *grid infrastructure*, *grid application* and *fault tolerance management*. Table 7.1 shows the list of parameters, the type and possible values of each parameter, and in which experiment each parameter is used.

7.1.1 Basic Parameters

The basic parameters of the testbed are *type of execution*, *failure seed*, *recovery seed* and *experiment size*. These parameters are needed in all experiments. The type of execution determines whether the grid application is executed in virtual-time or real-time. The failure and the recovery seeds, on the other hand, are integer numbers that initialize two random number generators, which we refer to as *failure generator* and *recovery generator*, respectively. The failure generator produces a sequence of numbers that are used to check whether a given activity fails or not. The recovery generator also produces a sequence of numbers that are used to check whether the failed activity can be recovered or not. The last basic parameter of the testbed, the experiment size, specifies the number of times an experiment is run, each run with unique failure and recovery seeds.

7.1.2 Grid Infrastructure Parameters

The parameters of the testbed that are related to the grid infrastructure are *number of nodes* and *number of cores per node*. These parameters are needed in all experiments.

7.1.3 Grid Application Parameters

The grid application parameters describe the structure and the behaviour of a grid application. Some of these parameters are shared by all grid applications. The rest are applicable to either MR or CL grid applications.

For all grid applications, the overhead of global communication and barrier synchronisation, and the probability of activity failure should be given. The *global communication overhead* is the time that is needed to transfer the output of one activity to another. The *barrier synchronisation overhead* is the total time that is needed to ensure the completion of a given superstep. The *probability of activity failure* is the likelihood of an activity of a grid application to fail.

The parameters of an MR grid application include *number of map activities*, *type of reduction*, *map activity execution time*, and *reduce activity execution time*. The number of map activities and the type of reduction should be given if an MR grid application is executed in virtual-time. The type of reduction notifies the testbed whether the map activities are reduced sequentially or hierarchically. Map or reduce activity execution time is the time that is needed to complete the execution of a map or a reduce activity without failing. We refer to such execution time as the *failure free execution time* of an activity.

The parameters of a CL grid application are *dependency structure*, *type of parallelism*, *number of sets of input streams*, *number of stream elements*, and *CL activity execution time*. The testbed uses the dependency structure of a CL grid application to identify how data flows between the activities of the application. The testbed expects the dependency structure to be expressed using the syntax of the DAG generator that is known as Task Graph for Free (TGFF) [Dick et al., 1998]. In order to identify the CL supersteps, the testbed also expects the user to explicitly state the type of parallelism that is used during the computation. The testbed recognises coarse-grain, pipeline and repeated parallelism. If repeated parallelism is used, the number of the sets of input streams is required. Despite input streams ideally never stopping, our evaluation assumes bounded length of streams. Therefore, the total number of stream elements should be given. The last parameter, CL activity execution time, is the failure free execution time of a CL activity.

7.1.4 Fault Tolerance Management Parameters

The fault tolerance management parameters describe the behaviour of the fault tolerance support in a RACS. Some parameters are needed in all experiments, while others only in specific scenarios.

The type of the RAC architecture, the type of fault tolerance strategy, the number of maximum retry, and the probability of unrecoverable failure are required in all experiments. The selection of a RAC architecture is made via the *type of RAC architecture* parameter. The testbed recognises generic, MR-specific and CL-specific RAC architectures. The selected RAC architecture can be combined with restart, checkpointing-rollback or replication fault tolerance strategy. The user specifies her preferred fault tolerance strategy via the *type of fault tolerance strategy* parameter. The *number of maximum retry* is the utmost number of attempts to either recover a failed activity or avert the impending failure of an activity by replication. The maximum number of retry in restart, replication and checkpointing fault tolerance strategies

is referred to as the maximum number of *restarts*, *replicas* and *rollbacks*, respectively. The *probability of unrecoverable failure* is the likelihood of a failed activity to never recover.

In a typical RAC architecture, the execution of a proactive strategy is based on a runtime prediction. Thus, unless the user selects either no or reactive only fault tolerance support; the *prediction interval*, the *prediction overhead*, the *probability of false positives*, and the *probability of false negatives* should be given. The prediction interval is the duration between two successive predictions, while the prediction overhead is the time that is needed to make a single prediction. The user also has an option to opt-out of using a runtime prediction.

The remaining fault tolerance management parameters are concerned with the overhead of a fault tolerance strategy. *Replica overhead, checkpointing cost, rollback cost* and *restart overhead* represent the time that is needed to initiate the execution of the replica of an activity, to save the current state of an activity, to restore a failed activity to a stable state, and to restart a failed activity, respectively.

7.2 Testbed In Action

An experiment run commences after the values of the input parameters, which are needed for the experiment run, are given. During the run, the testbed engages in the following tasks: *grid application execution*, *fault injection*, *failure prediction*, *fault tolerance management*, and *data collection*. Upon the completion of these tasks, the testbed outputs the simulated reliability, execution time and cost of the grid application execution.

7.2.1 Grid Application Execution

The testbed executes a grid application in either real or virtual-time. In a real-time execution, the testbed submits the application to Xgrid or Condor. In a virtual-time execution, the testbed advances the execution without actually running the activities of the grid application. Suppose the failure free execution time of an activity is x. Given the activity completes execution without failure, virtually executing this activity is equivalent to adding the activity's execution time to the total execution time of the application, i.e. $T = T + x$, where T is the execution time so far. The testbed also simulates the transfer of data between activities. This is done by adding the global communication overhead, denoted by y, to the execution time so far, i.e., $T = T + y$. The testbed assumes equal load distribution during virtual-time executions.

For a virtual-time execution, the testbed requires information about the grid infrastructure and the grid application. The number of nodes and the number of cores per node should

be given to the testbed regardless of the class of the grid application. If the class of the grid application is MapReduce, the testbed expects the number of map activities, the type of reduction, and the failure free execution times of map and reduce activities. Since the testbed assumes equal load distribution, the map activities will be equally distributed among the available cores before execution begins. Then, the cores concurrently execute their allocated map activities. If each core is allocated two or more map activities, the outputs of the map activities on each core will be locally reduced. Upon the completion of all of map activities and local reductions, the testbed commences global reductions, which are either sequential or hierarchical. When the global reduction is completed, so is the execution of the MR application.

If the class of the grid application is Combinational Logic, then the testbed expects the dependency structure, the type of parallelism, the number of stream elements, the number of the sets of input streams, and the failure free execution time of CL activities. The testbed uses the dependency structure and the type of parallelism to identify all CL supersteps of the computation. Since the testbed assumes bounded streams, the number of CL supersteps is finite. The testbed commences the CL computation by executing the activities in the first CL superstep. When these activities complete execution, the testbed then executes the activities in the next superstep. This continues until the execution of the activities in the last CL superstep is completed. Similar to MR activities, before the execution of activities in a CL superstep commences, the activities are equally distributed among available cores.

7.2.2 Fault Injection

The testbed injects real and simulated faults during the execution of a grid application. If the application is executed in real-time, then activities are randomly killed. If the application is executed in virtual-time, then simulated faults are injected to terminate virtually running activities. The testbed estimates the time of fault injection based on an exponential distribution. The execution time of an activity, which is killed by a simulated fault, is the total time the activity would be running until the time of the fault injection. The testbed injects faults at the node level. Thus, all activities that are running on the node, where the fault is injected, are assumed to fail.

7.2.3 Failure Prediction

The testbed regularly checks the presence of an impending failure. The testbed uses a random uniform distribution, along with the probability of false negatives and false positives, to de-

termine the presence of an impending failure before the next prediction. Further, the testbed uses the prediction interval parameter to determine the time to make the next prediction.

At the time of any prediction, there are two mutually exclusive events: *failure is impending* or *failure is not impending*. Given failure is impending, the testbed will either correctly predict the presence of a threat in the near future, i.e. true positive, or not, i.e., false negative. The prediction outcome under this scenario depends on the probability of false negatives. Given failure is not impending, the testbed will either recognize the absence of a threat in the near future, i.e., true negative, or send an unnecessary warning, i.e., false positive. The prediction outcome under this scenario depends on the probability of false positives.

In the testbed, a prediction is made at the node level. Therefore, whenever an impending failure is predicted, the testbed executes a proactive strategy to avert the impact of the impending failure on all activities that are running on the node. For example, if the proactive strategy is replication, then a replica of all activities will be instantiated.

7.2.4 Fault Tolerance Management

The testbed provides fault tolerance support to grid applications that are executed in virtual-time. The fault tolerance support is simulated, and thus the selected fault tolerance strategy is executed in virtual-time.

The testbed can be configured to follow the principles of the RAC approach as well as existing alternative fault tolerance approaches. For instance, instead of replicating an activity based on positive failure prediction, which is the case in the RAC approach, the testbed replicates all activities n times before the execution of the application commences. The testbed also offers an option to execute a grid application without any fault tolerance support. Further discussions focus on describing the behaviour of the testbed when the RAC-based fault tolerance support is provided.

The type of the fault tolerance support depends on the selected RAC architecture and fault tolerance strategy. Therefore, we identify each support using the type of the RAC architecture and the fault tolerance strategy with which the architecture is paired. For example, if the managers in MR-specific architecture handle failure using replication, then the fault tolerance support is referred to as the MR-specific replication-based RAC.

- **Restart-based RAC**: The restart-based RAC manages failure only reactively. The generic restart-based RAC restarts a failed activity whose computation does not depend on previously completed activities, whereas both the MR-specific restart-based RAC and

the CL-specific restart-based RAC can restart any failed MR and CL activity, respectively.

- **Replication-based RAC**: The replication-based RAC manages failure only proactively. If an activity is predicted to fail, then the replica of the activity will be executed. If the impending failure of the activity is not predicted prior to the activity's failure, no attempt is made to recover the activity. In the replication-based RAC, at most two replicas of an activity are simultaneously executed. If a positive prediction is made while the two replicas are being executed, no more replica is instantiated even if the maximum replica limit is not reached. The generic replication-based RAC can replicate an activity only if the activity does not depend on other activities, whereas the MR-specific replication-based RAC and the CL-specific replication-based RAC can replicate any MR and CL activity, respectively.

- **Checkpointing-based RAC**: The checkpointing-based RAC manages failure proactively and reactively. The checkpointing-based RAC saves the current state of an activity whenever the activity is predicted to fail. If/when the activity fails, the activity is rolled-back to the last checkpoint. Unlike the restart and the replication counterparts, the generic checkpointing-based RAC can recover the failure of *any* type of activity provided that specific conditions are met. If an activity that depends on a previously completed activity fails *and* the activity is checkpointed before its failure, then the generic checkpointing-based RAC recovers the failed activity. The MR-specific checkpointing-based RAC and the CL-specific checkpointing-based RAC can, with no restriction, manage the failure of any MR and CL activity, respectively.

The testbed restarts, replicates and rolls-back an activity a fixed number of times. If the activity is unable to successfully complete its execution after the maximum retry limit is reached, the activity is considered to fail beyond recovery. All other activities that depend on the failed activity will be marked as failed beyond recovery. The testbed can also be configured to execute a fault tolerance strategy an unlimited number of times. However, such configuration should be used cautiously as there is a possibility for an activity to continuously fail and never be able to successfully complete its execution.

The testbed uses the probability of unrecoverable failure along with a random uniform distribution to determine whether a failed activity can be recovered or not. The testbed performs this type of check only in the restart-based RAC and the checkpointing-based RAC.

Since the restart-based RAC and the checkpointing-based RAC attempt to recover a failed activity, it is essential to determine whether the failed activity is recoverable or not. If the failure is unrecoverable, then no attempt is made to recover the activity.

7.2.5 Data Collection

While the execution of a grid application is in progress, the testbed records the execution time and the cost of the application so far. At the end of each execution, the testbed outputs the status of the execution (completed or failed), the execution time and the cost of the application. The status of a grid application execution is used to determine the reliability of the application under the given conditions. In Sections 7.2.5.1, 7.2.5.2 and 7.2.5.3, we discuss how the testbed estimates the reliability, the total execution time and the cost of a grid application execution, respectively.

7.2.5.1 Reliability

Every experiment is run multiple times with unique failure and recovery seeds. At the end of each run, the testbed records the status of the execution. In line with the discussion in Section 5.2.1, all of the activities of a grid application must successfully complete in order for the execution of the application to be regarded as a success. Otherwise, if any of the activities fails beyond recovery, then the execution is considered as a failure. Once the multiple executions of a given experiment are completed, the testbed reports the reliability of the grid application execution under the given constraints according to Equation (7.1).

$$Reliability = \frac{\text{Number of successfully completed runs}}{\text{Number of all runs}} \tag{7.1}$$

The reliability estimation of a CL grid application is slightly different from what is discussed above. During a CL computation, *each* experiment run involves *multiple* executions of a CL application. In each CL experiment, a CL application is executed as many times as the total number of the stream elements. Therefore, when an experiment run is completed, the testbed records the number of the stream elements that are successfully processed. Then, when all of the experiment runs are completed, the testbed reports the reliability of CL application according to Equation (7.2), provided that n be the total number of experiment runs, m be the total number of stream elements, and x_i be the number of successfully processed stream elements in experiment run i.

$$Reliability = \frac{\sum_i^n x_i}{n \times m} \tag{7.2}$$

7.2.5.2 Execution Time

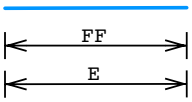
The testbed either records or estimates the execution time of a grid application. If the application is executed in real-time, then the execution time is recorded. Otherwise, the execution time is estimated. Hereafter, our discussion focusses on estimating the execution time of a grid application.

The execution time of a grid application depends on the architecture of the grid application, the execution time of the application's activities, the number of available cores, the type of the scheduler, and the data transfer overhead. The testbed assumes that the distribution of the initial input data and executable files to the cores is already taken care of. Therefore, the testbed does not include the time that is needed to distribute these files in the total execution time. The testbed also assumes the scheduling policy to be equal load distribution. Before presenting how the execution time of a MapReduce and a Combinational Logic grid applications are estimated in Sections 7.2.5.2.2 and 7.2.5.2.3, Section 7.2.5.2.1 discusses the estimation of the execution time of an activity under various scenarios.

7.2.5.2.1 Execution Time of an Activity

The execution time of an activity depends on what happens to the activity and/or its execution environment during execution. In this section, we present four selected scenarios that show how the testbed estimates activity execution time. The execution time of an activity in all other scenarios can be constructed from the discussions below. When the execution time of an activity is estimated, the testbed does not include the duration between activity failure and failure detection in the total execution time of the activity.

Scenario 1 (Failure Free Execution). Suppose an activity completes execution without failing. The execution time of the activity is the same as its failure free execution time, Figure 7.1.

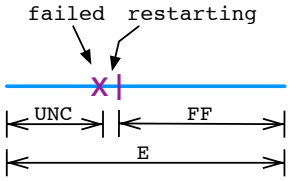


Legend `E`: total execution, `FF`: failure free execution

Figure 7.1: Execution of an activity without failing

Scenario 2 (Execution with Restart). Suppose an activity fails, and then successfully completes execution after restart. The total execution time of such activity is equal to the time

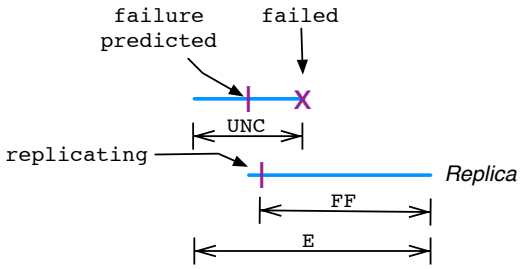
the activity spent on an uncompleted execution, the restarting overhead, and the failure free execution time of the activity, Figure 7.2.



Legend E: total execution, FF: failure free execution, UNC: uncompleted execution

Figure 7.2: Execution of an activity with restart

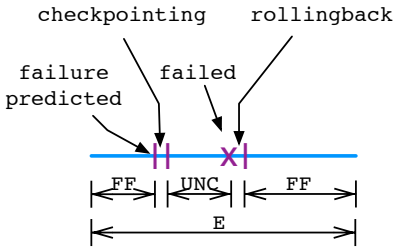
Scenario 3 (Execution with Replication). Suppose the impending failure of an activity is predicted, the activity is replicated before its failure, and the replica of the activity is successfully executed. The execution time of such activity is the sum of the activity's failure free execution time, the replication overhead, and the time the activity spent on the original execution before the failure prediction, Figure 7.3. Though the original execution continues until it fails, the duration between failure prediction and failure occurrence in the original execution does not contribute towards the total execution time of the activity. This is because the computation in this duration overlaps with the execution of the activity's replica.



Legend E: total execution, FF: failure free execution, UNC: uncompleted execution

Figure 7.3: Execution of an activity with replication

Scenario 4 (Execution with Checkpointing-Rollback). Suppose an activity is checkpointed before its failure, and then the activity successfully completes execution after rollback. The total execution time of such activity is the sum of the failure free execution time of the activity, the time the activity spent on computing after the checkpoint but before its failure (uncompleted execution), and the time that is needed to checkpoint and rollback the activity Figure 7.4.



Legend E: total execution, FF: failure free execution, UNC: uncompleted execution

Figure 7.4: Execution of an activity with checkpointing-rollback

7.2.5.2.2 Execution Time of a MapReduce Application

The execution time of a MapReduce application is the time that is needed to complete the map phase with optional local reductions, the transfer of data between activities, and a global reduction. The execution time of the map phase is determined by the core whose allocated map activities take the longest time to complete and be locally reduced.

The total data transfer time from either map or local reduce activities to global reduce activities, or from one global reduce activity to another in hierarchical reduction depends on the type of the communication channel. If the output data can be sent in parallel, the data transfer time is $g \times \log_2 c$; where g is the global communication overhead and c is the number of cores that are used during the map phase. If the communication channel is shared, however, the total data transfer time will be $g \times (c - 1)$. The testbed assumes shared communication channel.

The execution time of a global reduction depends on its type, i.e., sequential or hierarchical. The execution time of a sequential global reduction is the sum of the execution times of all global reduce activities. For a hierarchical global reduction, the execution time is the sum of the execution time of the longest reduce activity in each level (see Figure 6.1). Suppose c cores are used during the map phase, the map activities are reduced in $\log_2 c$ levels.

7.2.5.2.3 Execution Time of a Combinational Logic Computation

As discussed in Section 7.2.1, the testbed identifies all CL supersteps of a CL computation. Since a CL computation is a sequence of CL supersteps, the testbed adapts Valiant's cost model, as shown in Equation (4.1), to estimate the execution time of a CL computation. The execution of CL activities in a CL superstep is equivalent to local computation in Valiant's

superstep. Transferring the outputs of these activities to the activities in the next superstep represents Valiant's global communication. Finally, the time the testbed adds at the end of global communications, before executing activities in the next CL superstep, is equivalent to Valiant's barrier synchronisation.

The adaptation of Valiant's model is needed to incorporate the assumption of the testbed about communication channel being shared, and to include scenarios where the number of CL activities in a superstep is greater than the number of allocated cores for the computation. The adapted Valiant's model is shown in Equation (7.3).

Let

- S: the total execution time of a CL computation
- s_i: the execution time of CL superstep i
- $w_{i,j,k}$: the execution time of CL activity j in superstep i on core k
- h_j: the h-relation of activity j
- g: the transmission capacity of the network to deliver data (bandwidth)
- b: fixed (amortised) cost of synchronisation (conceptual barrier)

$$S = \sum_{i=1}^{m} s_i \qquad \text{for } m \text{ total CL supersteps in } S \tag{7.3}$$

$$s_i = \max_{k=1}^{c} \Big(\sum_{j=1}^{n/c} w_{i,j,k} \Big) + \sum_{j=1}^{n} (h_j \times g) + b \tag{7.4}$$

$$\text{where } c: \text{ total cores allocated for CL superstep } i$$

$$n: \text{ total CL activities in superstep } i$$

7.2.5.3 Cost

The cost of a grid application execution is the total CPU time that is spent on executing the activities of the application, fault tolerance management and prediction.

We use the cost of a grid application, instead of the total execution time, for our reliability-overhead analysis. The user of a grid service is either allocated a specific resource usage quota if the service is free, or is billed based on the amount of resources she uses to execute her

application, which is standard especially in service providers like GoGrid [GoGrid Website] and Amazon [Amazon Website]. Therefore, since the total execution time only reflects the length of the computation, we chose the cost of a grid application, which shows the area of the computation. The area of the computation, as represented by time, is the total computing power that is consumed by the application and the fault tolerance support.

Suppose p be the cost of the execution of a grid application without fault tolerance support, and q be the cost of the execution of a grid application with fault tolerance support. The unscaled overhead of the fault tolerance support is $q - p$. For the reliability-overhead analysis, we normalize the unscaled overhead according to Equation (7.5).

$$Normalised\ Overhead = \frac{q}{p} - 1 \tag{7.5}$$

7.3 Summary

In this chapter, we introduced our parameterised experiment testbed that is designed to empirically evaluate a given RAC architecture with respect to reliability and overhead. The testbed can be configured in many ways through its grid application, grid infrastructure and fault tolerance management parameters. During an experiment run, the testbed engages in grid application execution, fault injection, failure prediction, fault tolerance management, and data collection. Upon the completion of these tasks, the testbed outputs the simulated reliability, execution time and cost of the grid application execution.

Chapter 8

Experiment Design

The objective of our experiment runs is to answer our research questions (Section 1.1). Therefore, we design various sets of experiments to analyse

i. the reliability-overhead tradeoff that is enabled by the generic, the MR-specific and the CL-specific RAC architectures (Sections 9.1.1 and 9.1.2),

ii. the impact of the parameters of a fault tolerance strategy, which is paired with a given RAC architecture, on the tradeoff (Sections 9.2 and 9.3), and

iii. the sensitivity of the tradeoff to prediction interval (Section 9.4), and false negative and false positive predictions (Section 9.5).

For each set of experiments, we assign a single value or a range of values to the parameters of the testbed. As shown in Table 7.1, we recognise the testbed parameters as *controlled* and *independent* variables. The value of a controlled variable is held constant in all experiments, while the value of an independent variable is changed from one experiment to another. In all experiments, the *dependent* variables* are reliability and cost. The experimental setup of the controlled and independent variables is discussed in Sections 8.1 and 8.2, respectively.

Our experiments are based on simulation. We aim to evaluate the RAC approach for a range of independent variables settings, where the value of each independent variable is uncertain and the impact of an independent variable is defined not necessarily in isolation but in its complex interactions with others. Real data for such combinations of conditions are hard to get by, and if

*Generally in a scientific experiment, the experiment aims to establish the dependency of certain observations, which are the dependent variables; and the factors that these observations depend upon, which are the controlled and the independent variables [Welkowitz et al., 2012].

we measure them in a real cluster or grid within a real application run or even thousands of runs, these 'real' contexts are still only few samples in the sea of so many applications, frameworks and architectures combinations that we are interested in. Consequently, how well the 'real' data represents the part of the 'sea' we are interested in, is entirely uncertain. Likewise, when we look at dependent variables in these 'real' runs, it remains very uncertain, what kind of generalisations we can make about the dependency measured. Therefore, simulation is the only option we have in general for this type of wide-ranging problem [Zurell et al., 2010].

8.1 Controlled Variables

The results of all experiments depend on *type of execution, failure seed, recovery seed, experiment size, number of nodes, number of cores per node, global communication overhead, barrier synchronisation overhead*, and *probability of activity failure*. These are the shared controlled variables. The remaining controlled variables in Table 7.1 are specific to either MR or CL related experiments; therefore we discuss these variables separately in Sections 8.1.1 and 8.1.2, respectively.

For all experiment runs, a grid application execution and fault tolerance management are carried out in virtual-time. In order to minimise noise from the data, an experiment run is repeated 10^3 times, each time with unique failure and recovery seeds. We allocate 32 dual-core compute nodes to each experiment run, and thus, upto 64 activities can be executed in parallel.

The overheads of global communication and barrier synchronisation are obtained from the Pallas MPI Benchmarks (PMB) suite [PMB, 2000]. The *Ping Pong* and the *Barrier* benchmarks of the PMB suite are used to measure the overhead of global communication and barrier synchronisation, respectively. We run the benchmarks on our HPC cluster of 32 nodes with 8 cores each. The global communication overhead depends on the size of the data to be transferred, while the barrier synchronisation overhead depends on the number of activities to be synchronised. For example, the global communication overhead of transferring 2MB of data is 8.45μsec, and the barrier synchronisation overhead for synchronising 64 activities is 638.52μsec.

We assume individual activities of a grid application to be highly reliable, and thus we limit the probability of activity failure within the range of $\{0, 0.005, 0.01, 0.015, \ldots, 0.2\}$. The fact that failure in a grid application execution being highly likely, when compared to non-grid applications, does not necessarily imply that the failure probability of the individual activities of the application is very high. The high likelihood of failure in a grid is due to factors like the

massive number of grid components, which have non-zero probability of failure, the longevity of grid computations, and the like (see Section 4.3.1 for details).

8.1.1 MapReduce Experiments

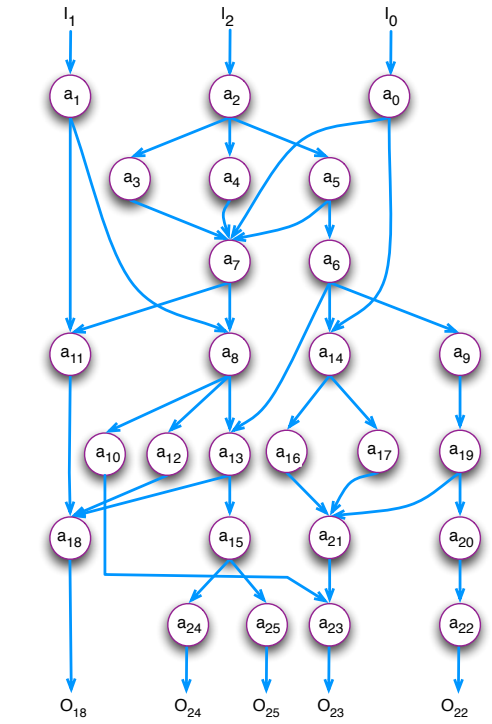
The controlled variables that are needed in MR experiments are *number of map activities, map execution time, reduce execution time,* and *type of global reduction.*

Our simulated MR grid application is inspired by 4CareK [Peake et al., 2009], a parameterised large scale distributed lattice generator. 4CareK allows the user to specify, among other things, the number of map activities and the execution time of each activity via its parameters. For our experiment, we assume a simulated 4CareK grid application with 256 map activities; therefore, each of the 64 cores will be allocated four activities. Since the execution time of each map activity of 4CareK can be configured, we assume each map activity to take 15 minutes in one set of experiments and 1 hour in another. Inspired by the MPI_REDUCE routine and 4CareK, we assume global sequential reduction. We measured the execution time for reducing the map activities of 4CareK on our Xgrid. The execution time is roughly 1500 msec, which is very small. Therefore, in order to study MR grid applications with expensive reduce activities, we also set the execution time of each reduce activity to be the same as and significantly more than the execution time of a map activity.

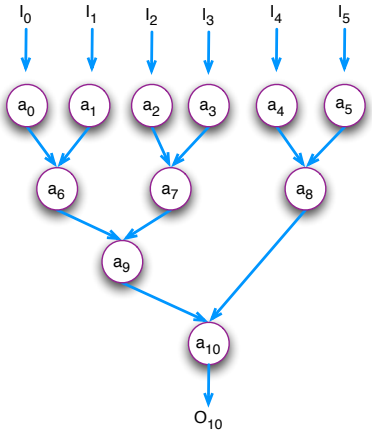
8.1.2 Combinational Logic Experiments

The controlled variables that are needed in CL experiments are *dependency structure, type of parallelism, number of sets of input streams, number of stream elements,* and *CL activity execution time.*

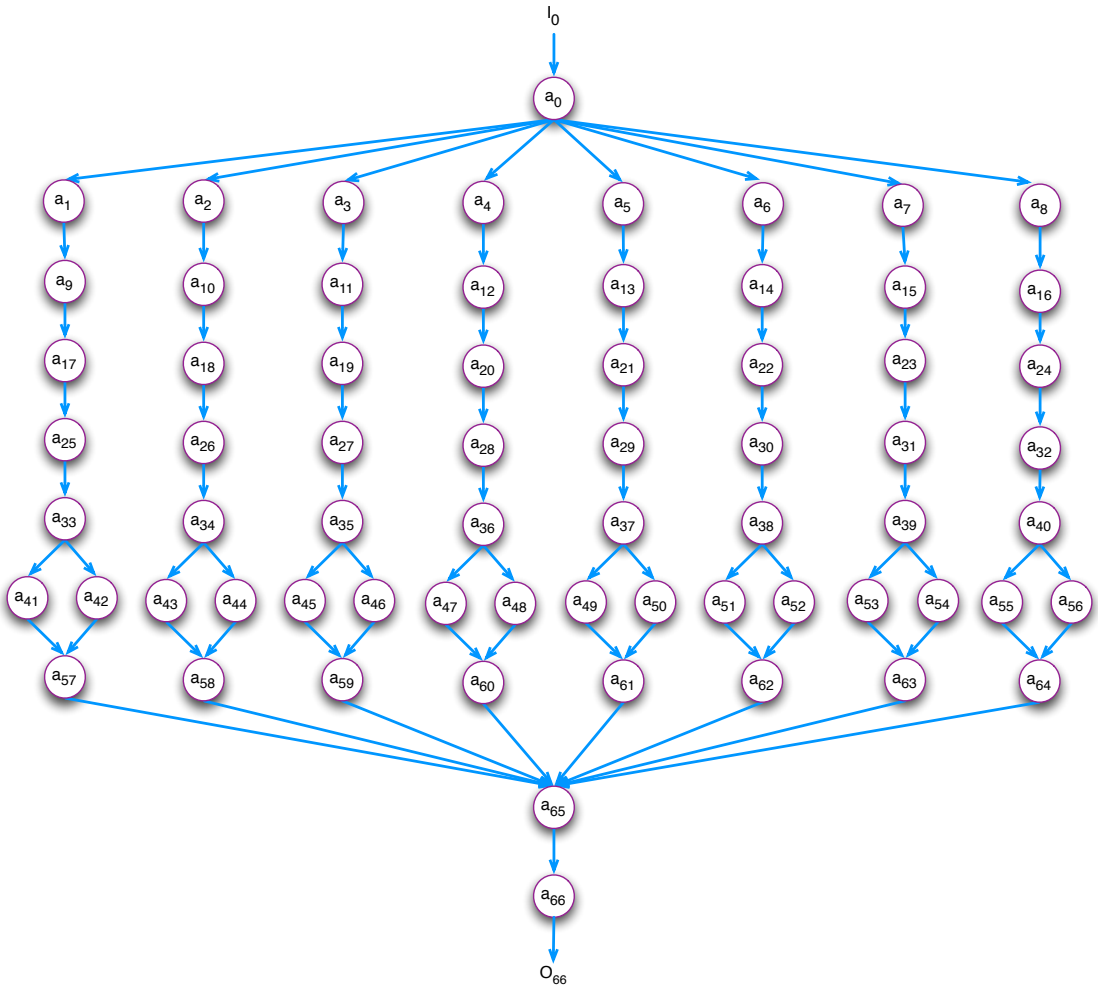
We execute a randomly generated and two real CL applications during CL experiment runs. We refer to the randomly generated application as *Tgff* since the application is generated by the TGFF tool (Section 7.1.3). Tgff has 26 CL activities and 7 processing steps. The other CL applications are Spatial Matching [Kuntschke et al., 2006], an astrophysics application that is required to determine Spectral Energy Distributions (SEDs), and FilterBank [Gordon, 2010], a multi-rate signal decomposer in, for example, image processing. Spatial Matching has 11 CL activities and 4 processing steps, while FilterBank has 67 CL activities and 10 processing steps. The DAGs of Tgff, Spatial Matching and FilterBank are shown in Figure 8.1. We assume pipeline and repeated parallelism during the execution of these CL applications. The number of input stream sets is 2, while the number of stream elements is 50. In-line with our assumption



(a) Tgff



(b) Spatial Matching



(c) FilterBank

Figure 8.1: Benchmark DAGs for CL Experiments

in Section 6.2.1.2, each CL activity of a CL computation is assigned equal execution times. For each CL computation, the execution time of its CL activities are randomly generated. The randomly generated execution times represent small and long execution times, in their respective experiment runs.

8.2 Independent Variables

The independent variables that are needed in all experiments are *type of RAC architecture, type of fault tolerance strategy, number of maximum retry,* and *probability of unrecoverable failure.* The rest of the independent variables are related to prediction, and the overhead of the selected fault tolerance strategy. We discuss the values of these variables in Sections 8.2.1 and 8.2.2.

In MR experiments, the fault tolerance support is based on the generic and the MR-specific RAC architectures. Likewise, in CL experiments, we use the generic and the CL-specific RAC architectures. Each RAC architecture is paired with restart, replication or checkpointing-rollback fault tolerance strategy. These strategies represent reactive only, proactive only and hybrid failure handling. As discussed in Section 3.3.1, there are other fault tolerance strategies that a RAC architecture could be paired with; however, such strategies are represented by the selected strategies at an abstract level. Redundancy, standby spare and N-version are similar to replication; despite the difference in the actual implementation, all of them advocate the use of more than one instance of an activity to manage failure. The reason checkpointing is combined with rollback rather than with migration is that checkpointing-rollback is a hybrid strategy while checkpointing-migration is a proactive strategy; a proactive strategy is already represented by replication. Rejuvenation is also not considered due to its purely proactive nature.

We leave the number of maximum retries unbounded and determine the bound (if any) by experiment. We assume a non-zero probability for an activity to never recover after its failure. Possible causes of an unrecoverable failure include internal software bug, data corruption during execution, memory leak, CPU crash, and others. Though there are multiple events that could trigger such failure, they are not necessarily frequent. Therefore, we set the probability of unrecoverable failure to be low, i.e. 0.01. Later, in Section 9.2, we study the general impact of the probability of unrecoverable failure on the performance of a RAC architecture. For such study, the values of the probability of unrecoverable failure are 0, 0.01, 0.05, 0.1, 0.3, and 0.75.

8.2.1 Prediction Variables

The independent variables that are specific to prediction are *prediction interval, probability of false negatives, probability of false positives,* and *prediction overhead.* We discuss the prediction overhead, along with other fault tolerance overheads, in Section 8.2.2.

8.2.1.1 Prediction Interval

We set the prediction interval to be 5% of the failure free execution time of an activity. In MR experiments, since map and reduce activities could have different execution times, the MR-specific RAC architecture adjusts the prediction interval for each type of activity. Suppose the execution time of a map and reduce activity is 2000 and 50, respectively. During the map phase, a prediction is made every 100 units of time, while in the reduce phase, a prediction is made every 2.5 units of time. The generic RAC architecture does not change the prediction interval as the architecture assumes all activities to be the same for the purpose of fault tolerance support. Later, in Section 9.4, we study the general impact of the prediction interval on the performance of a RAC architecture. For such study, the values of the prediction interval are 5%, 15%, 25%, ..., 95% of the failure free execution time of an activity.

Note that, in the checkpointing-based RAC, the prediction interval should be greater than the overhead of a single checkpoint. Suppose positive predictions are made during the n^{th} and $n+1^{st}$ predictions, for some natural number n. If the prediction interval is less than the overhead of a single checkpoint, then the second checkpointing begins while the first is in progress. Since the activity to be checkpointed is still suspended when the second checkpointing begins, the second checkpointing does not save an advanced state of the activity. Such scenario can be avoided by using a prediction interval that is larger than the overhead of a single checkpoint.

8.2.1.2 Prediction Accuracy

The probabilities of false positives and false negatives determine the accuracy of a predictor. In an ideal predictor, which knows the state of the system and makes perfect prediction all the time, these probabilities are zero. In the other extreme, a predictor will have no access to the state of the system, and therefore, 'guesses' the current status of the system by, for example, tossing a coin. We refer to such a predictor as *state oblivious.* All other predictors lie between these two. These predictors have limited or full access to the system. However, since

"*prediction is very difficult, especially about the future*"[†], their predictions are not necessarily correct all the time. We refer to these predictors as *state aware*. An ideal predictor is a state aware predictor that makes perfect predictions.

We use state oblivious predictors to set the baseline for the quality of state aware predictors. Since state aware predictors take into account the current condition of the execution environment, these predictors must be more accurate than the state oblivious ones. Otherwise, it is not worth putting any effort to develop these predictors. In all experiments, except where stated otherwise, we assume a state oblivious predictor. This is because we would like to study how a RAC architecture performs despite the available predictor being the least accurate one. Our state oblivious predictor is randomised, i.e., the predictor makes a prediction based on a random constant k, where $k \in [0, 1]$. k is the probability of making positive prediction regardless of the current status of the execution environment with respect to an impending failure. At the time of prediction, this predictor generates a random number i, where $i \in [0, 1)$, and then compares i to k. If $k \geq i$, then a positive prediction is made. Otherwise, a negative prediction is made.

In our experiment testbed, k cannot be directly used. The testbed expects the probability distribution of a predictor's outcomes to be expressed by the probabilities of false positives and false negatives (Section 7.1.4). However, the probability distribution of a state oblivious predictor is given in terms of positive and negative predictions. Even though there are two mutually exclusive events at the time of prediction (Section 7.2.3), *failure is impending* and *failure is not impending*, a state oblivious predictor does not intrinsically recognise these events. Therefore, the probability of positive predictions remains the same irrespective of the current event. Following is summarised the probability distribution of a state oblivious predictor.

- Given an impending failure at the time of prediction, the probability of true positives is k and the probability of false negatives is $1 - k$.
- Given no impending failure at the time of prediction, the probability of true negatives is $1 - k$ and the probability of false positives is k.

Let the probability of false positives given no impending failure be x, and the probability false negatives given an impending failure be y. From the probability distribution of a state oblivious predictor, we observe that $x + y = 1$. In our experiment, $k = 0.5$, and thus, the probabilities of both false positives and false negatives are 0.5. Later, in Section 9.5, we will

[†]The source of the quote is disputed. However, we attribute the quote to Niels Bohr.

discuss how the performance of a RAC architecture is affected by the accuracy of a state oblivious predictor, where $k \in [0, 1]$, and a state aware predicator.

8.2.2 Overhead Variables

The independent variables that are related to overhead are *checkpointing cost, rollback cost, replica overhead, restart overhead,* and *prediction overhead.*

For our evaluation, we assume system level checkpointing, and therefore measure the time that is needed to take a memory dump. The checkpointing cost of a 4CareK activity on NFS is 131 msec, and the size of the checkpoint is 2MB. Since the overhead of checkpointing an activity depends on the complexity of the activity, the size of the information to be saved, the location of the checkpointing storage (local disk vs. NFS), and the network bandwidth [Plank et al., 1999], we therefore study the impact of the cost of checkpointing on checkpointing-based RAC in Section 9.3 in detail. According to Nurmi et al. [2005], the overheads of checkpointing and rollback are roughly the same in long running jobs on Condor. Therefore, we assign equal values for the rollback cost and the checkpointing cost.

We measure restart and replication overheads. Since restart and replication create a new instance of an activity, they both need the same input values. In 4CareK, for example, the overhead of reading the input values of an activity from NFS is 0.66 msec. We also measure the overhead of making prediction by a state oblivious predictor, and it is 0.014 msec. All our measurements are done using YourKit Java Profiler [YourKit Website].

8.3 Experiment Runs Presentation

For each evaluation, we only show the most significant portions of the data, and elide other redundant portions that show essentially the same thing. The outputs of all experiment runs will be provided on request.

Figure 8.2 shows the reference graph notation, which subsequent graphs in this thesis will follow. Each graph is divided into two, each half representing reliability and overhead. For the value of a given variable in the x-axis, the reliability of a grid application whose failure is managed by a specific fault tolerance support type is plotted on the *Reliability* half, and the overhead of the fault tolerance support is shown on the *Overhead* half.

The graph has an additional dimension based on color to represent a fault tolerance support type. `RPL-g`, `CHK-g` and `RST-g` denote generic replication-based RAC, checkpointing-based RAC and restart-based RAC, respectively; and their respective architecture-specific equivalents

are `RPL-s`, `CHK-s` and `RST-s`. Finally, `NoFT` shows a grid application execution without any fault tolerance support.

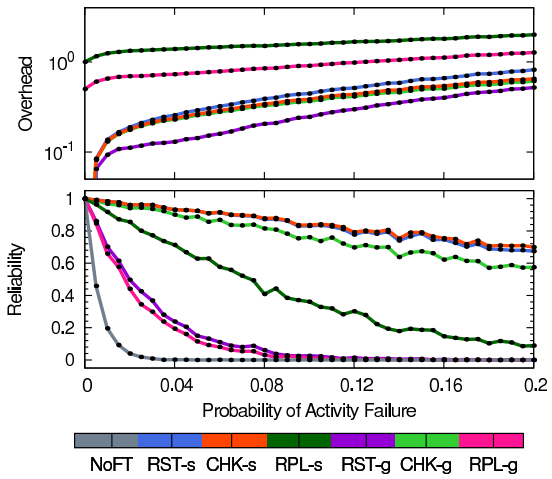


Figure 8.2: Reliability-overhead tradeoffs from one of the benchmark runs.

8.4 Summary

In this chapter, we presented the design of our experiment. We described the values of the controlled and the independent variables in Table 7.1. Finally, we provided a reference graph notation that will be used throughout the thesis.

Chapter 9

Results

In this chapter, we answer our research questions (Section 1.1): *What is the reliability-overhead tradeoff that is enabled by the RAC approach for MapReduce and Combinational Logic grid applications? How sensitive is such reliability-overhead tradeoff to a fault tolerance strategy and its parameters, and prediction accuracy?* We answer the first question by presenting the reliability-overhead tradeoff that is enabled by the generic, the MR-specific and the CL-specific RAC architectures. We then answer the second question by discussing the impact of an unrecoverable failure on the reliability-overhead tradeoff, how the cost of checkpointing affects the performance of the checkpointing-based RAC, and the sensitivity of the checkpointing-based and the replication-based RAC to prediction interval and accuracy. These discussions are based on the scenario presented in Figure 9.1c. Nonetheless, the knowledge is transferable to other scenarios as well.

9.1 Reliability and Overhead under RAC Architecture

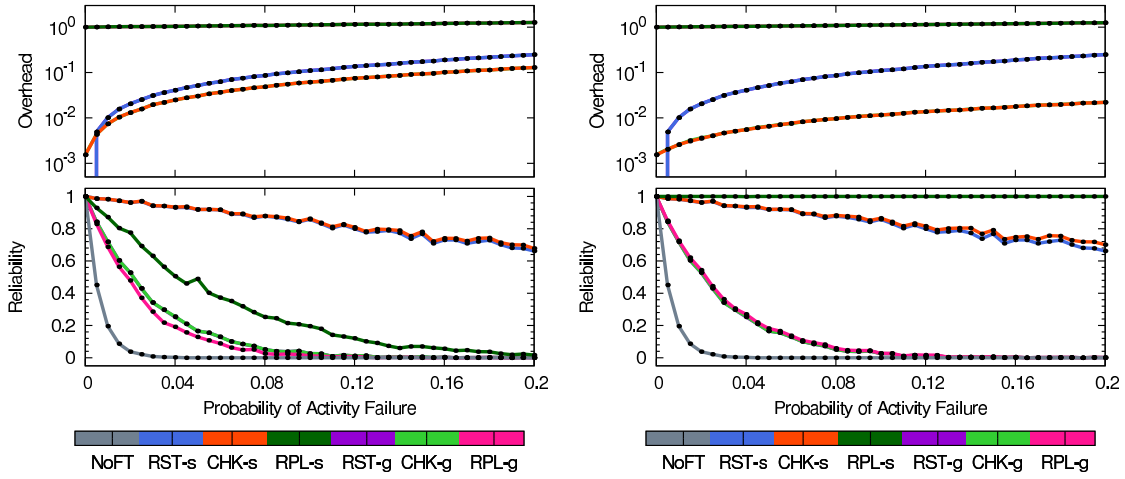
The generic and the architecture-specific RAC approaches improve the reliability of MR and CL grid applications. Such reliability improvement, nevertheless, comes at the expense of increased cost of execution. The extent of the reliability improvement and the overhead of providing such improvement depend on the type of the RAC architecture (generic, MR-specific, CL-specific), and the fault tolerance strategy with which the RAC architecture is paired. We present the reliability improvement that MR and CL grid applications would gain by adapting the RAC approach and the associated cost of the reliability improvement in Sections 9.1.1 and 9.1.2.

9.1.1 The Case of MapReduce

Under the default parameter settings, the execution of an MR grid application without fault tolerance support almost always leads to failure at application level. In such situations, the reliability of the execution is almost zero if the probability of activity failure is greater than 0.04. This is shown in Figure 9.1. According to our reliability definition (Section 7.2.5.1), the failure of one activity suffices to consider the entire execution as failed. Combining either generic or MR-specific fault tolerance support with an MR execution improves the reliability of the execution up to a limit.

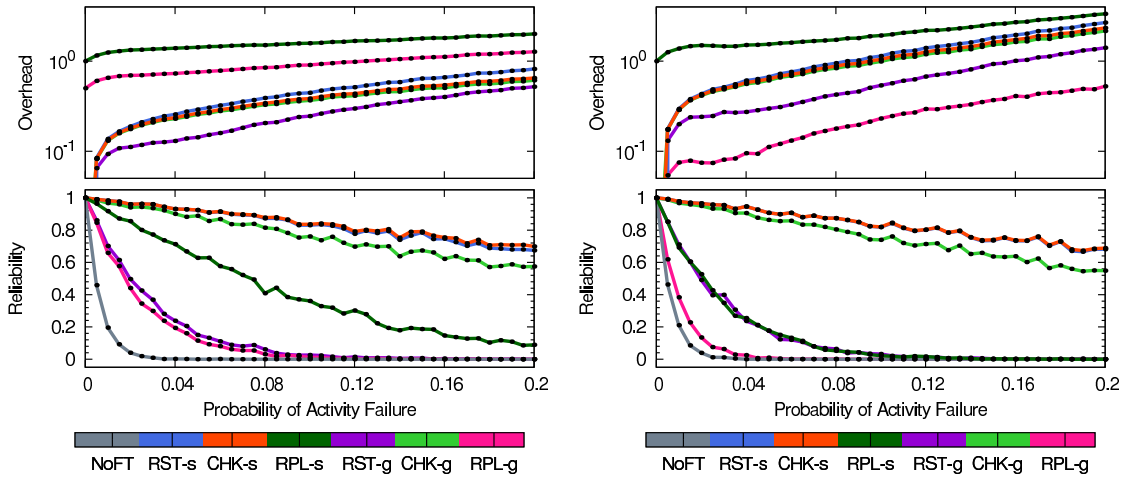
We evaluated the reliability-overhead tradeoff of the generic and the MR-specific RAC architectures for MR grid applications. In summary,

- *Result 1*: MR-specific fault tolerance support provides a more reliable MR application execution than generic fault tolerance support. Figure 9.1.
- *Result 2*: Given the execution time of a map activity is significantly higher than the execution time of a reduce activity, the overhead of MR-specific fault tolerance support is almost the same as the overhead of its generic counterpart. Figures 9.1a and 9.1b.
- *Result 3*: Given the execution time of a reduce activity is at least equal to the execution time of a map activity, the overhead of MR-specific fault tolerance support, with the exception of the checkpointing-based RAC, is significantly higher than the overhead of generic fault tolerance support. The generic and the MR-specific variants of the checkpointing-based RAC incur roughly the same overhead. Figures 9.1c and 9.1d.
- *Result 4*: The MR-specific restart-based and the MR-specific checkpointing-based RAC provide almost equally reliable execution of an MR grid application. Figure 9.1.
- *Result 5*: Given the failure of some activities requires reactive fault tolerance management, the MR-specific restart-based and the MR-specific checkpointing-based RAC provide a more reliable execution of an MR grid application than any of the other fault tolerance support types. Figures 9.1a, 9.1c and 9.1d.
- *Result 6*: Given a proactive strategy can be executed prior to the failure of any activity and the cause of the failure is a transient fault, the MR-specific replication-based RAC provides the most reliable MR computation. Figure 9.1b.



(a) Long running map and short running reduce. Failure at any time.

(b) Long running map and short running reduce. Failure towards the end.



(c) Equal execution time for map and reduce. Failure at any time.

(d) Short running map and long running reduce. Failure at any time.

Figure 9.1: The reliability-overhead tradeoff of the generic and the MR-specific RAC.

- *Result 7*: Given the probability of activity failure is non-zero, of the MR-specific fault tolerance support types, the replication-based RAC introduces the highest overhead, followed by the restart-based RAC and then the checkpointing-based RAC. Figure 9.1.

9.1.1.1 Restart-Based RAC

The MR-specific restart-based RAC provides a more reliable execution of an MR gird application than the generic restart-based RAC, Figure 9.1. This is due is to the fact that the MR-specific restart-based RAC can manage the failure of both map and reduce activities, whereas the fault tolerance support by the generic restart-based RAC is limited to map activities. The inability of the generic restart-based RAC to recover a failed reduce activity makes it provide a less reliable execution of an MR grid application than its MR-specific counterpart.

The superiority of the MR-specific restart-based RAC over the generic restart-based RAC becomes more pronounced than before as the probability of activity failure increases. As shown in Figure 9.1a, for example, if the probability of activity failure is 0.01, the generic restart-based RAC and the MR-specific restart-based RAC guarantee a reliability of 0.72 and 0.98, respectively. In this case, the MR-specific RAC provides 0.26 more reliability than the generic RAC. However, if the probability of activity failure becomes 0.03, the difference in the reliability of the computation reaches 0.6; while the MR-specific RAC achieves 0.94 reliability, the generic RAC guarantees only 0.34 reliability. The key reason behind the widening of the gap with an increase in the probability of activity failure is that as the probability of activity failure increases, the likelihood of the reduce activity to fail increases as well. Since the generic restart-based RAC does not have a mechanism to deal with reduce failures, such increase highly impacts its performance.

The MR-specific restart-based RAC introduces more overhead than its generic counterpart. Nonetheless, the degree of the overhead difference depends on the relative execution time of reduce and map activities. When the execution time of a map activity is considerably higher than the execution time of the reduce activity, as shown in Figures 9.1a and 9.1b, the overhead of the MR-specific restart-based RAC is only marginally larger than the overhead of the generic restart-based RAC. In this scenario, the majority of the overhead is incurred during the map phase. Since both types of fault tolerance support put equivalent effort to handle the failure of map activities, they introduce more or less comparable overhead. The additional attempt to restart failed reduce activities by the MR-specific restart-based RAC will not introduce significant overhead as the execution time of the reduce activities is very small, especially when compared to the execution time of the map activities.

In the scenario when the execution time of a reduce activity is at least the same as the execution time of a map activity, as shown in Figures 9.1c and 9.1d, the overhead of the MR-specific restart-based RAC is significantly higher than the overhead of the generic restart-

based RAC. Since only the MR-specific restart-based RAC attempts to recover a failed reduce activity, which in this case is very expensive, we observe a significant gap between the overhead of the two types of fault tolerance support.

9.1.1.2 Replication-Based RAC

Both the generic and the MR-specific variants of the replication-based RAC improve the reliability of an MR grid application up to a limit. The performance of these types of fault tolerance support is limited by the lack of any mechanism to recover a failed activity. This, as shown in Figures 9.1a, 9.1c and 9.1d, becomes apparent as the probability of activity failure increases. With an increase in the probability of activity failure, more and more activities will fail before their impending failure can be predicted, all other things kept constant. If the failure of activities is not predicted, then no replica will be instantiated.

Despite an increase in the probability of activity failure, if a proactive strategy can be executed prior to the failure of any activity and the cause of the failure is a transient fault, then the MR-specific replication-based RAC guarantees a 100% reliable computation. Under the default parameter settings, if an activity fails only towards the end of its execution, a positive prediction will be made at some point during its computation, and consequently the activity will be replicated. Such execution is shown in Figure 9.1b. As long as the cause of the failure is a transient fault and a replica is instantiated before the failure of the activity, the activity eventually completes successfully. If the cause of the failure is a permanent fault, regardless of how many times the activity is replicated, it will not complete successfully. The generic replication-based RAC does not guarantee a 100% reliable computation as its performance is limited by its inability to not only recover a failed activity but also initiate a replica of a reduce activity.

The relationship between the generic and the MR-specific variants of the replication-based RAC with respect to overhead is similar to that of the restart-based RAC. Given the execution time of a map activity is considerably higher than the execution time of a reduce activity, as shown in Figures 9.1a and 9.1b, the MR-specific replication-based RAC introduces almost equivalent overhead as the generic replication-based RAC. However, if the execution time of the reduce activity is at least the same as the execution time of the map activity, as shown on Figures 9.1c and 9.1d, then the overhead of the MR-specific replication-based RAC becomes significantly higher than that of its generic counterpart.

9.1.1.3 Checkpointing-Based RAC

The MR-specific checkpointing-based RAC provides a more reliable execution of an MR grid application than the generic checkpointing-based RAC, Figure 9.1. The generic and the MR-specific variants of the checkpointing-based RAC save the current state of both map and reduce activities after a positive prediction. The difference between the two comes at the time of rolling-back a reduce activity. If the reduce activity is not checkpointed before its failure, unlike the MR-specific checkpointing-based RAC, the generic checkpointing-based RAC cannot recover it. This is why the MR-specific checkpointing-based RAC provides a more reliable computation than its generic counterpart.

The extent of the reliability of an MR computation, whose failure is managed by the generic checkpointing-based RAC, depends on the relative execution time of map and reduce activities. In the scenario when the execution time of a map activity is considerably higher than that of a reduce activity, the reliability of the computation is low. Otherwise, the reliability of the computation is relatively high. This is due to the way the prediction interval is configured during the execution of an MR grid application.

The prediction interval in the generic checkpointing-based RAC is set up with respect to the activity that has the *longest* execution time. Suppose the respective execution time of a map and a reduce activity are 10^4 and 5 units of time, and the prediction interval be 5% of the failure free execution time of an activity. Thus, the prediction interval will be 500. Since, such prediction interval is larger than the execution time of the reduce activity, no prediction will be made during the execution of a reduce activity; and consequently none of the reduce activities will be checkpointed. In this scenario, the reliability of the MR grid application, whose fault tolerance support is provided by the generic checkpointing-based RAC, will be low due to the inability of the generic RAC to recover a failed reduce activity that is not checkpointed. This scenario is shown in Figures 9.1a and 9.1b.

In the scenario when the execution time of a reduce activity is higher than that of a map activity, the performance of the generic checkpointing-based RAC will not be affected by the prediction interval. By reversing our previous example, let the execution time of a map and a reduce activity be 5 and 10^4, respectively. Since the longest activity execution time is used to set the prediction interval, the prediction will still be 500. Due to the large difference between the execution time of a map activity and the prediction interval, no prediction will be made during the map phase. Nonetheless, since the generic checkpointing-based RAC can recover a failed map activity even if the activity is not checkpointed, the performance of the

generic checkpointing-based RAC will not be affected due to the prediction interval setting. In such scenario, as shown in Figure 9.1d, the generic and the MR-specific variants of the checkpointing-based RAC provide a comparably reliable MR computation.

The extent of the reliability of an MR computation, whose failure is managed by the MR-specific checkpointing-based RAC, does not depend on the relative execution time of map and reduce activities. This is because the prediction interval in the MR-specific checkpointing-based RAC is set up based on the execution time of *each activity type*. In the previous example, where the execution time of a map activity is 10^4 and the execution time of the reduce activity is 5, the prediction interval is 500 during the map phase and 0.25 during the reduce phase. Therefore, there will be prediction during the execution of all activities.

The overhead of the MR-specific checkpointing-based RAC is marginally higher than that of the generic checkpointing-based RAC irrespective of the relative execution time of a map and a reduce activities. As discussed in Section 9.1.1.2, if the execution time of the map activity is significantly higher than that of the reduce activity, the bulk of the overhead comes from the map phase. As the result, both fault tolerance support types introduce marginally the same overhead. If the execution time of the map activity is not significantly higher than that of the reduce activity or if the execution time of the reduce activity is at least the same as that of the map activity, the overhead comes from both the map and the reduce phases. Since such scenario leads to favourable prediction interval settings to the generic checkpointing-based RAC, the generic checkpointing-based RAC puts almost as equal effort as its MR-specific counterpart to handle the failure of both map and reduce activities. As the result, both fault tolerance support types introduce comparable overhead.

9.1.1.4 Comparing the RAC-based Fault Tolerance Support Types

In Sections 9.1.1.1-9.1.1.3, we established that MR-specific fault tolerance support provides a more reliable execution of MR grid applications than generic fault tolerance support. Therefore, in this section, we will focus on only the MR-specific fault tolerance support types. The overall relationship among the MR-specific fault tolerance support types is as follows:

- The MR-specific variants of the restart-based and the checkpointing-based RAC provide almost equally reliable execution of an MR grid application. This is because, unless an activity fails due to an unrecoverable failure, both fault tolerance support types repeatedly restart/rollback the activity until it completes successfully. Despite the similarity in behaviour with respect to repeated retries, the MR-specific checkpointing-based RAC

sometimes provides a more reliable computation than the MR-specific restart-based RAC. Since a rolled-back activity generally has shorter execution time than a restarted activity, the rolled-back activity stands a better chance of avoiding an unrecoverable failure, and subsequently completing successfully than the restarted one. The longer the execution time, the more likely to fail.

- The MR-specific restart-based and the MR-specific checkpointing-based RAC provide a more reliable computation than the MR-specific replication-based RAC, provided that the failure of some activities requires reactive fault tolerance management. This is due to the fact that the MR-specific replication-based RAC being a purely proactive fault tolerance support. However, there are scenarios in which the MR-specific replication-based RAC provides equally or more reliable computations than its checkpointing and restart counterparts. Examples of such scenarios include when the probability of unrecoverable failure is more than 0.05 (Section 9.2), when the probability of false positives is more than 0.8 (Section 9.5), and when the predictor is perfect.

- The MR-specific replication-based RAC introduces the highest overhead of all. Since a replica is instantiated due to both true and false positive predictions, the MR-specific replication-based RAC almost always doubles the cost of the execution. The next costly fault tolerance support is the MR-specific restart-based RAC. Whenever an activity fails, the computation is started from the beginning. In fact, the MR-specific restart-based RAC is equivalent to the MR-specific replication-based RAC, provided that the MR-specific replication-based RAC replicates an activity based on only true positive predictions. Since the MR-specific replication-based RAC unnecessarily replicates an activity because of false positive predictions, it is more costly than its restart counterpart. Under the default parameter settings, given the probability of activity failure is non-zero, the MR-specific checkpointing-based RAC introduces the least overhead. Since the computation that will be lost is the one from the last checkpoint, it is expected for the MR-specific checkpointing-based RAC to be relatively less costly than the others. This is especially evident from Figure 9.1b, on which the overhead of the checkpointing-based RAC is shown to be almost one order of magnitude less than the restart-based RAC. When the probability of activity failure is zero, unnecessary checkpointings due to false positive predictions make the MR-specific checkpointing-based RAC introduce more overhead than the MR-specific restart-based RAC.

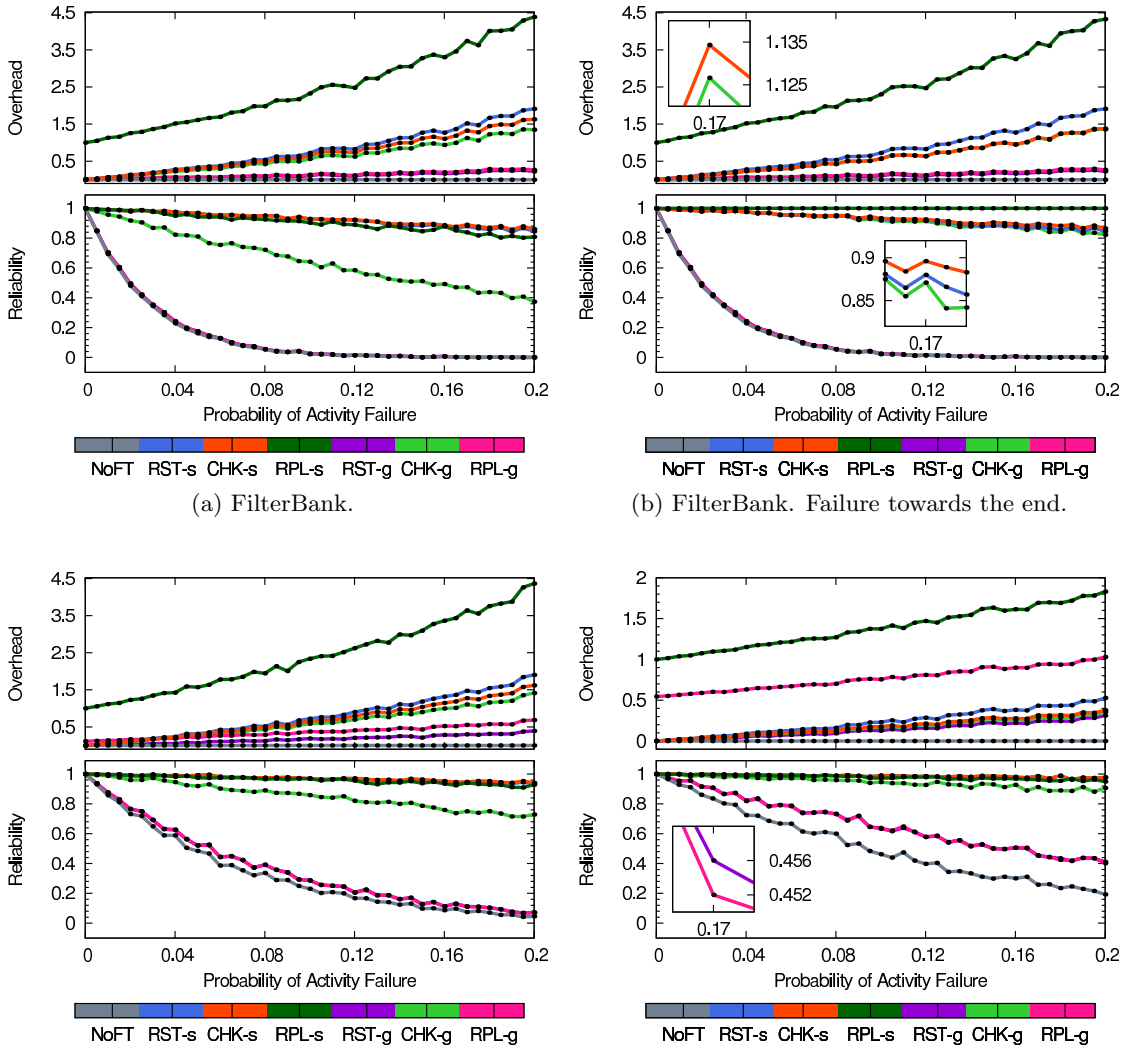
9.1.2 The Case of Combinational Logic

We evaluated the reliability-overhead tradeoff of the generic and the CL-specific RAC architectures for CL applications. In summary:

- *Result 8*: CL-specific fault tolerance support provides a more reliable execution of a CL application than generic fault tolerance support. Figure 9.2.
- *Result 9*: The overhead of CL-specific fault tolerance support, with the exception of the checkpointing-based RAC, is notably higher than the overhead of generic fault tolerance support. Figure 9.2.
- *Result 10*: Given a proactive strategy can be executed prior to the failure of any activity, the generic and the CL-specific variants of the checkpointing-based RAC provides almost equally reliable computation of a CL application. Figure 9.2b.
- *Result 11*: The CL-specific restart-based and the CL-specific checkpointing-based RAC provide almost equally reliable execution of a CL grid application. Figure 9.2.
- *Result 12*: Given the failure of some activities requires reactive fault tolerance management, the CL-specific restart-based and the CL-specific checkpointing-based RAC provide a more reliable execution of a CL grid application than any of the other fault tolerance support types. Figure 9.2a.
- *Result 13*: Given a proactive strategy can be executed prior to the failure of any activity and the cause of the failure is a transient fault, the CL-specific replication-based RAC guarantees 100% reliable execution. Figure 9.2b.
- *Result 14*: Given the probability of activity failure is non-zero, of the CL-specific fault tolerance support types, the replication-based RAC introduces the highest overhead, followed by the restart-based RAC and then the checkpointing-based RAC. Figure 9.2.

9.1.2.1 Restart-Based RAC

The CL-specific restart-based and the generic restart-based RAC improve the reliability of a CL application execution. The CL-specific restart-based RAC provides, as shown in Figure 9.2, a more reliable execution of a CL grid application than the generic restart-based RAC. This is the result of the CL-specific RAC being able to handle the failure of any activity, and the



(a) FilterBank.

(b) FilterBank. Failure towards the end.

(c) Tgff.

(d) Spatial Matching.

Figure 9.2: The reliability-overhead tradeoff of the generic and the CL-specific RAC: The inset figures magnify selected data to show the relationship between the plotted fault tolerance support types whose plots are overlapped on the scale of the outer figure.

inability of the generic RAC to manage the failure of an activity whose execution depends on previously completed computations. Hereafter we refer to an activity whose computation depends on previously completed activities as a *successor activity*.

The superiority of the CL-specific restart-based RAC over its generic counterpart becomes more pronounced than before with an increase in the probability of activity failure, and/or the *complexity* of a given CL application, i.e., the size and the degree of communication among the

activities of the application. As the probability of activity failure increases, the likelihood of a successor activity to fail increases as well. This disadvantages the generic restart-based RAC as the generic RAC does not have a mechanism to deal with the failure of a successor activity. As the complexity of a CL application increases, so does the number of successor activities, e.g., Spatial Matching vs. Tgff. The more the successor activities, the less the degree of the fault tolerance support by the generic restart-based RAC.

The CL-specific restart-based RAC introduces more overhead than its generic counterpart. The extent of the difference in overhead depends on the complexity of a given CL application. As the complexity of a CL application increases, so does the overhead difference between the two variants of the restart-based RAC. Since the generic restart-based RAC does not handle the failure of successor activities, its overhead is limited to the management of CL activities in the first processing step only; whereas the overhead of the CL-specific restart-based RAC is incurred in all processing steps as the CL-specific RAC can manage the failure of any CL activity. The overhead difference between the generic RAC and the CL-specific RAC, for instance, is significantly higher in Tgff than in Spatial Matching. This is expected since Tgff has more activities, more communication among its activities and more processing steps than Spatial Matching. In Tgff, as shown in Figure 9.2c, the increase in the cost of execution is up to 175% by the CL-specific RAC and only 50% by the generic RAC. However, in Spatial Matching, as shown in Figure 9.2d, we observe a relatively small gap between the two. The increase in the cost of execution is up to 50% by the CL-specific RAC and 30% by the generic RAC.

9.1.2.2 Replication-Based RAC

The CL-specific replication-based and the generic replication-based RAC improve the reliability of a CL application execution. However, their performance is limited by their inability to recover a failed activity. With an increase in the probability of activity failure, as shown in Figures 9.2a, 9.2c, and 9.2d, more and more activities fail before their impending failure can be predicted. Unless a prediction is made, the replica of an activity will not be instantiated. Despite an increase in the probability of activity failure, as shown in Figure 9.2b, if a proactive strategy can be executed prior to the failure of any CL activity and the cause of the failure is a transient fault, the CL-specific replication-based RAC guarantees a 100% reliable computation. Such behaviour is also shared by the MR-specific replication-based RAC, and therefore see Section 9.1.1.2 for further discussion, all reference to *reduce activity* and *MR* shall be understood to mean *successor activity* and *CL*, respectively.

The CL-specific replication-based provides a more reliable CL computation, and introduces a higher overhead than the generic replication-based RAC. The relationship between these fault tolerance support types is similar to the one between the generic and the CL-specific variants of the restart-based RAC. Therefore, see Section 9.1.2.1 for further discussion.

9.1.2.3 Checkpointing-Based RAC

Both variants of the checkpointing-based RAC improve the reliability of a CL application execution. The CL-specific checkpointing-based RAC generally provides a more reliable CL computation than the generic checkpointing-based RAC. This is due to the inability of the generic checkpointing-based RAC to recover a successor activity that was not checkpointed before its failure. However, if the activities of a CL application fail only towards the end of their computation or if the application is not complex, then both fault tolerance support types provide almost equally reliable CL computations.

In the case when a successor activity fails only towards the end of the computation, the likelihood of the activity to have been checkpointed is high. Given an activity fails after completing 95% of its computation, under the default parameter settings, where the prediction interval is 5% of a CL activity execution time and the probability of positive predictions is 0.5, there will be 19 predictions before the activity fails. Roughly half of these predictions will be positive, and therefore cause the activity to be checkpointed. Once a successor activity is checkpointed, the generic checkpointing-based RAC can manage its failure.

The complexity of a CL application is a good indicator of the extent of the presence of successor activities in the application. As the complexity of a CL application increases, from Spatial Matching to FilterBank, the number of successor activities in the application increases as well. The more complex the application is, the less manageable its failure will be by the generic checkpointing-based RAC, and vice versa.

The overhead of the CL-specific checkpointing-based RAC is marginally higher than the overhead of the generic checkpointing-based RAC, even when the reliability gap between the two is significant. Figure 9.2a, for example, shows that as the probability of activity failure increases, the difference between the two fault tolerance support types with respect to reliability increases at a faster speed than with respect to overhead. As long as an activity is checkpointed, the generic and the CL-specific RAC put equivalent effort to handle its failure. Under the default parameter settings, many of the activities of the benchmark CL applications are checkpointed more often than not, and thus we observe marginally equal overhead. How-

ever, due to the non-zero probability of false negative predictions, there are successor activities that will fail before they can be checkpointed. As discussed previously, the presence of such activities deteriorates the overall reliability of the application whose failure is managed by the generic checkpointing-based RAC.

9.1.2.4 Comparing the RAC-based Fault Tolerance Support Types

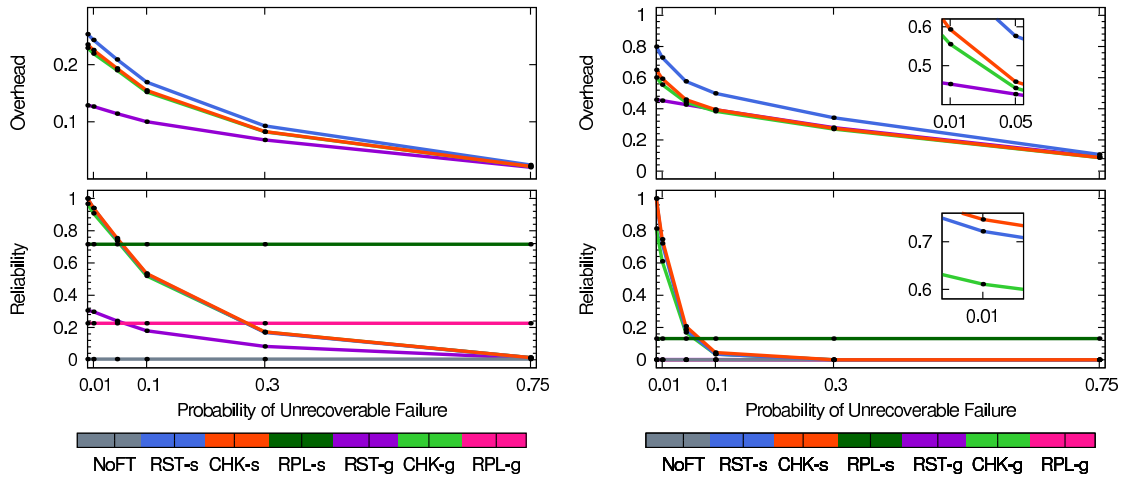
See Section 9.1.1.4, all reference to *MR* shall be understood to mean *CL*.

9.2 Probability of Unrecoverable Failure

The probability of unrecoverable failure is one of the independent variables that affects the performance of the restart-based and the checkpointing-based RAC. In summary:

- *Result 15*: As the probability of unrecoverable failure increases, the restart-based and the checkpointing-based RAC provide less and less reliable execution of an MR grid application, all other things being equal. The reliability of the computation eventually becomes smaller than the execution of the application with the replication-based RAC. Figure 9.3.
- *Result 16*: With an increase in the probability of unrecoverable failure, the overhead of the restart-based and the checkpointing-based RAC becomes less and less, all other things being equal. Figure 9.3.
- *Result 17*: Given the cause of the unrecoverable failure is not an internal software fault, an increase in the probability of unrecoverable failure does not have a notable impact on the replication-based RAC, all other things being equal.

The restart-based and the checkpointing-based RAC perform at a peak level when the probability of unrecoverable failure is 0. As the probability of unrecoverable failure increases, the performance of these fault tolerance support types deteriorates, all other things being equal. For instance, as shown in Figure 9.3, if the probability of unrecoverable failure is more than 0.05, the restart-based and the checkpointing-based RAC provide less reliable computation than the MR-specific replication-based RAC. By the time the probability of unrecoverable failure reaches 0.3, the restart-based and the checkpointing-based RAC will at best be as good as the generic replication-based RAC. The *minimum value* of the probability of unrecoverable failure that makes the restart-based and the checkpointing-based RAC perform less than the



(a) Probability of activity failure 0.035 (b) Probability of activity failure 0.175

Figure 9.3: The impact of the probability of unrecoverable failure on the reliability-overhead tradeoff: The overhead of the replication-based RAC is not plotted. The overhead is already shown in Figure 9.1c to be significantly higher than the other fault tolerance support types. Including such overhead in the plot obscures the impact of the probability of unrecoverable failure on the overhead of the restart-based and the checkpointing-based RAC, which is negligible for the replication-based RAC. The inset figures magnify selected data to show the relationship between the plotted fault tolerance support types whose plots are overlapped on the scale of the outer figure.

replication-based RAC is application and execution environment dependant. For example, as shown in Figure 9.3, for the probability of activity failure 0.035 and 0.175, the respective minimum value is 0.3 and 0.1.

With an increase in the probability of unrecoverable failure, as shown in Figure 9.3, the overhead of the restart-based and the checkpointing-based RAC decreases. Since no attempt is made to recover an activity that has failed beyond recovery, (Section 7.2.4), the more activities fail beyond recovery, the less time is spent on executing a fault tolerance strategy.

The probability of unrecoverable failure does not have a notable impact on the performance of the replication-based RAC, provided that the cause of the failure is not an internal software fault. When a replica of an activity is initiated due to a true positive prediction, the new replica is assumed to execute in a new environment. Despite the failure of the original activity beyond recovery, the event that triggered the unrecoverable failure on the original activity might not exist in the new environment. Thus, the new replica stands a good chance of terminating successfully. The restart-based and the checkpointing-based RAC, on the other hand, attempt to recover (restart/rollback) the failed activity on the same environment. Thus, no matter how

many times the attempt is made, the activity will keep failing.

9.3 Cost of Checkpointing

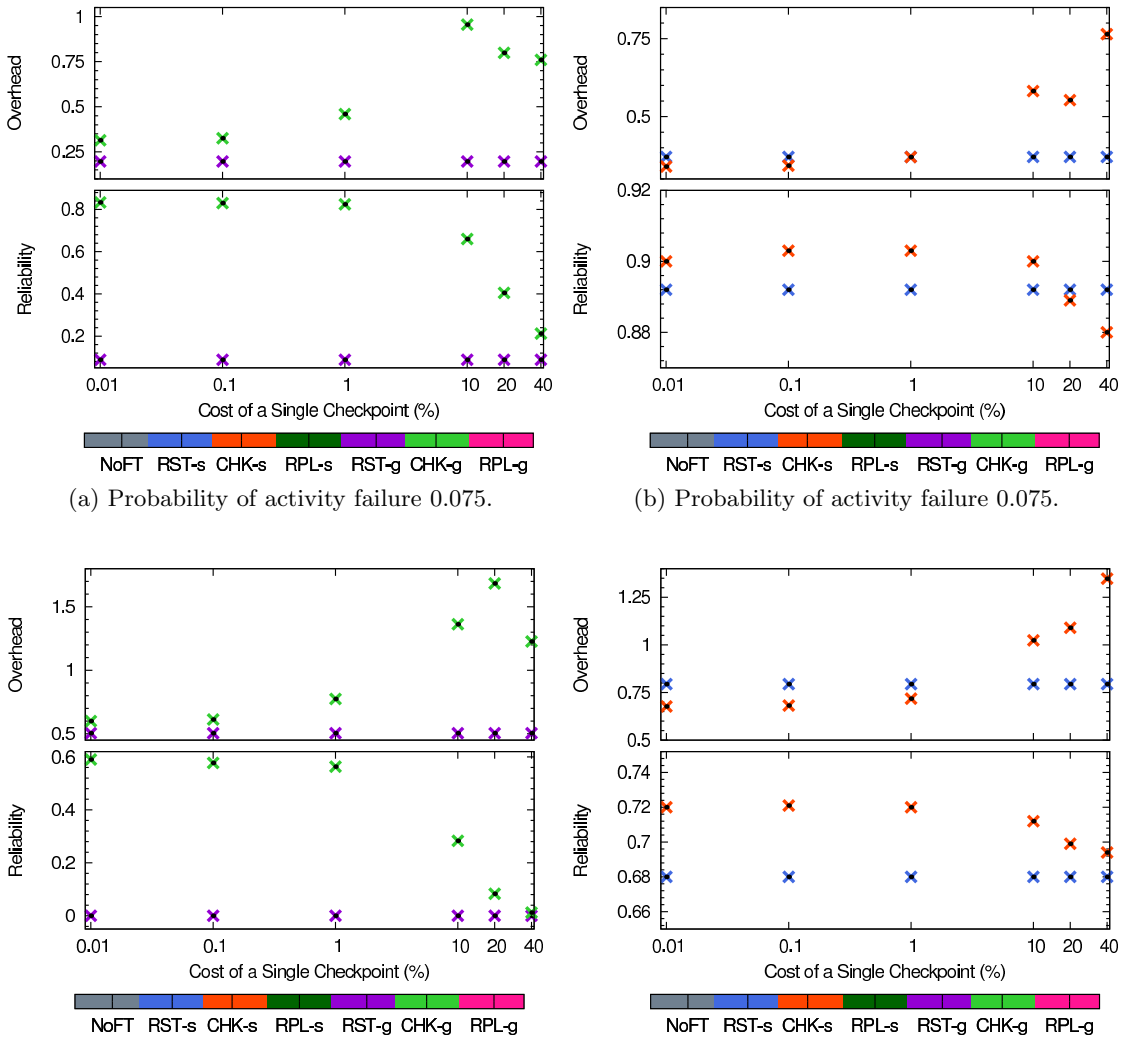
In this section, we evaluate the impact of the cost of checkpointing on the reliability-overhead tradeoff that is enabled by the checkpointing-based RAC. For such evaluation, we set the cost of checkpointing to be 0.01%, 0.1%, 1%, 10%, 20%, and 40% of the failure free activity execution time. Due to the requirement that the prediction interval must be greater than the cost of a single checkpoint (Section 8.2.1.1), we need to adjust the value of the current prediction interval, which is 5% of activity execution time, for some of the experiment runs. The current prediction interval is valid only for experiment runs whose cost of checkpointing is less than 5% of activity execution time. Since a prediction interval that is valid for a given experiment could be invalid for another, instead of fixing the prediction interval in each experiment run, we use a range of values. In each experiment run, we vary the prediction interval between 5% and 95% of activity execution time. Then, for the given cost of checkpointing, we choose the prediction interval that enables the checkpointing-based RAC to provide the most reliable computation with the least overhead possible. In summary:

- *Result 18*: As the cost of a single checkpoint increases, the reliability of the computation whose failure is managed by the checkpointing-based RAC either stays the same or decreases. Figure 9.4.

- *Result 19*: Provided that the reliability of a grid application computation remains almost constant, an increase in the cost of a single checkpoint generally increases the overhead of the checkpointing-based RAC. Table 9.1.

As the cost of a single checkpoint increases, the reliability of a grid computation either stays the same or decreases, all other things being equal. If an increase in the cost of a single checkpoint does not require to increase the prediction interval, then the reliability of the computation will be almost constant. This is because if there is no change in the prediction interval, the level of the proactive fault tolerance support provision would remain the same. For instance, Table 9.1 shows the absence of a change in the prediction interval despite an increase in the cost of a single checkpoint from 0.01% to 1% of activity execution time. This is why the reliability of the computation remains at approximately 0.83.

In the scenario when the increase in the cost of checkpointing necessitates a change in the prediction interval, we observe a significant drop in the reliability of the computation.



(a) Probability of activity failure 0.075.

(b) Probability of activity failure 0.075.

(c) Probability of activity failure 0.195.

(d) Probability of failure 0.195.

Figure 9.4: The impact of the cost of a single checkpoint on the reliability-overhead tradeoff of the checkpointing-based RAC.

Increasing the prediction interval implies more spread out predictions than before. Therefore, unless the predictor is highly accurate and can predict the status of the computation over an extended period of time, there is a good chance to overlook the presence of an impending failure before the next prediction point. This increases the number of activities that fail before being checkpointed. As the result, the performance of the checkpointing-based RAC will be affected. The performance degradation is more severe on the generic than on the MR-specific RAC due to the inability of the generic checkpointing-based RAC to recover an uncheckpointed

successor activity. For example, as shown in Table 9.1, if the cost of a single checkpoint is 10% of activity execution time, the prediction interval will be increased to 15% of activity execution time, and consequently the reliability of the computation will drop from 0.83 to 0.6.

Table 9.1: The impact of the cost of a single checkpoint on the generic checkpointing-based RAC. Probability of activity failure is 0.075. Equivalent data is shown in Figure 9.4a

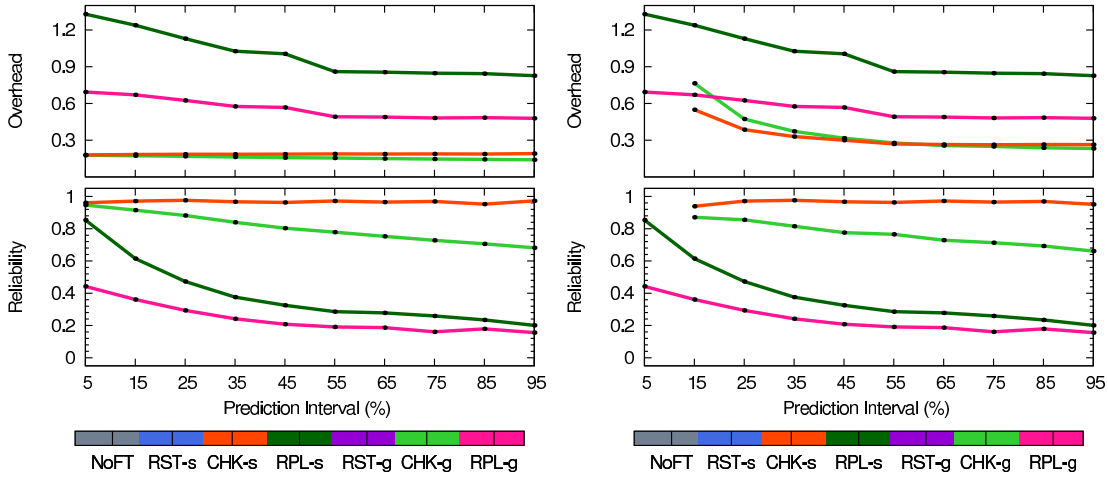
Cost of a single checkpoint	Prediction Interval	Reliability	Overhead
0.01%	5%	0.833	0.31
0.1%	5%	0.83	0.33
1%	5%	0.824	0.46
10%	15%	0.66	0.96
20%	35%	0.41	0.8
40%	75%	0.21	0.76

An expensive checkpointing strategy may make the checkpointing-based RAC provide less reliable computation than the restart-based RAC. Figure 9.4b, for example, shows that when the cost of a single checkpoint is 20% and 40% of activity execution time, the reliability of the computation is higher with the MR-specific restart-based RAC than with the MR-specific checkpointing-based RAC. Whenever a positive prediction is made, the execution time of the activity is extended by the time that is needed to complete the checkpoint (Section 7.2.5.2). Expensive checkpointing mechanisms make the execution time of the activity be significantly longer than before. The longer the execution time, the more likely the activity to fail due to an unrecoverable failure.

Overall, if the reliability of the computation is not changed due to the increase in the cost of a single checkpoint, the overhead of the checkpointing-based RAC generally increases. This is because the user is paying extra when she uses the expensive checkpointing without the benefit of increased reliability. Table 9.1 shows that when the cost of a single checkpoint increases from 0.01% to 1%, the reliability stays almost constant, but the overhead increases.

9.4 Prediction Interval

The prediction interval affects the reliability improvement of the replication-based and the checkpointing-based RAC offer to grid applications. In summary:



(a) Cost of a single checkpoint 0.01%.

(b) Cost of a single checkpoint 10%.

(c) Cost of a single checkpoint 20%.

(d) Cost of a single checkpoint 40%.

Figure 9.5: The impact of prediction interval on the reliability-overhead tradeoff of the checkpointing-based and the replication-based RAC. Probability of activity failure 0.02

- *Result 20*: With an increase in the prediction interval, the replication-based RAC provides less reliable computation and requires less overhead than before. Figure 9.5.
- *Result 21*: With an increase in the prediction interval, the checkpointing-based RAC increasingly provides more reliable computation than before, up to a limit. Once the limit is reached, the reliability remains roughly constant in the MR-specific checkpointing-based RAC, but decreases in the generic checkpointing-based RAC.

As discussed in Section 9.3, an increase in the prediction interval decreases the degree of proactive fault tolerance support provision, and increases the reliance on reactive strategies for failure management. As the result, an increase in the prediction interval affects the performance of fault tolerance support types that rely on proactive fault tolerance strategies to manage failure. The replication-based RAC and the generic checkpointing-based RAC are among these fault tolerance support types. As the prediction interval increases, since the replication-based RAC manages failure only proactively, its engagement in fault tolerance management becomes increasingly limited. As the result, it provides less reliable computation, and incurs less overhead than before.

In the checkpointing-based RAC, an increase in the prediction interval optionally increases reliability at first, and then the reliability stays roughly constant or decreases. This is shown in Figures 9.5c and 9.5d. An increase in the reliability of the computation is observed if the difference between the prediction interval and the cost of a single checkpoint is relatively small. In such configuration, whenever a positive prediction is made, a given activity completes only a fraction of its computation before the next prediction point. Consequently, a string of positive predictions significantly increase the execution time of the activity. The longer the execution time, the more chance to fail. Increasing the gap between the prediction interval and the cost of checkpointing enables activities to complete more computation between predictions than before, and subsequently shortens their execution time. Figure 9.5c, for example, shows that when the prediction interval is 25% of activity execution time, the generic checkpointing-based RAC guarantees 0.5 reliability. However, by increasing the prediction interval to 35% of activity execution time, up to 0.8 reliability can be achieved.

Once the prediction interval that guarantees the most reliable computation is achieved, which we refer to as the *optimal prediction interval*, increasing the prediction interval is neither necessary nor advisable practice. As a given prediction interval deviates from the optimal prediction interval, more and more activities will fail before being checkpointed. As discussed in Section 9.3, long prediction interval deteriorates the performance of the generic checkpointing-based RAC. Using longer prediction interval than the optimal one does not have a notable impact on the performance of the MR-specific checkpointing-based RAC. This is because the MR-specific checkpointing-based RAC can recover any activity, checkpointed or uncheckpointed, as long as the failure is recoverable.

9.5 Prediction Accuracy

The accuracy of a predictor, which is expressed by the probabilities of false positives and false negatives, influences the performance of the replication-based and the checkpointing-based RAC. Since the restart-based RAC does not use prediction for fault tolerance management, it is insensitive to the accuracy of a predictor. In this section, we discuss how the accuracy of a state oblivious predictor and a state aware predictor affect the reliability-overhead tradeoffs that are enabled by the replication-based and the checkpointing-based RAC.

9.5.1 State Oblivious Predictors

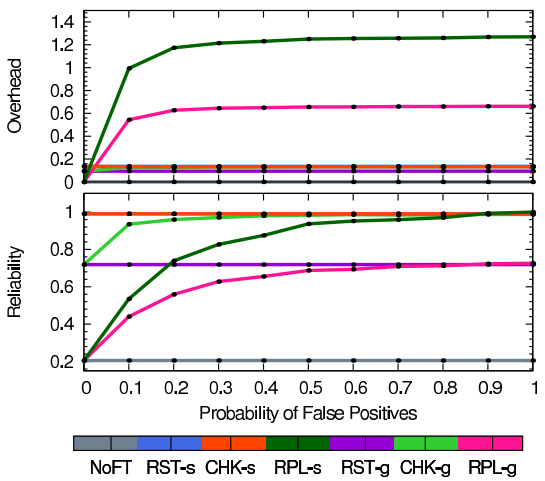
We discuss the impact of the accuracy of a state oblivious predictor based on a change in the probability of false positives only. The probability of false negative is $1 - x$, where x is the probability of false positives (Section 8.2.1.2). Therefore, studying the impact of an increase in false positive predictions is equivalent to studying the impact of a decrease in false negative predictions, and vice versa.

We evaluated the sensitivity of the reliability-overhead tradeoff to the probability of false positives of a state oblivious predictor. In summary,

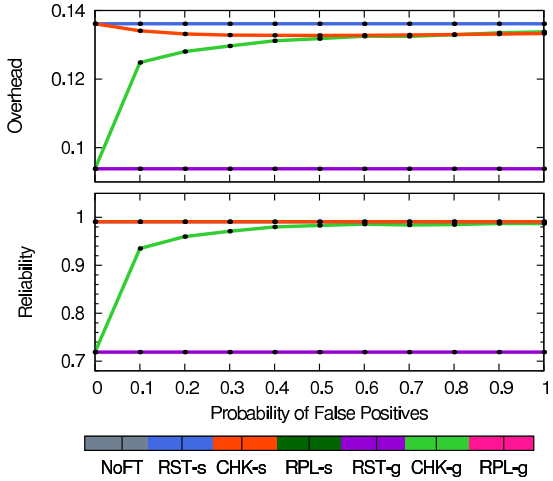
- *Result 22*: As the probability of false positives increases, the replication-based and the checkpointing-based RAC increase the reliability of an MR grid application execution up to a limit, all other things being equal. Figure 9.6.
- *Result 23*: With an increase in the probability of false positives, the overhead of the replication-based RAC and the generic checkpointing-based RAC increases, but the overhead of the MR-specific checkpointing-based RAC decreases up to a limit, all other things being equal. Figure 9.6.

Given failure is not impending, an increase in the probability of false positives increases the reliability that is provided by both replication-based and checkpointing-based RAC. The advantage of increasing the probability of false positives comes in two fold:

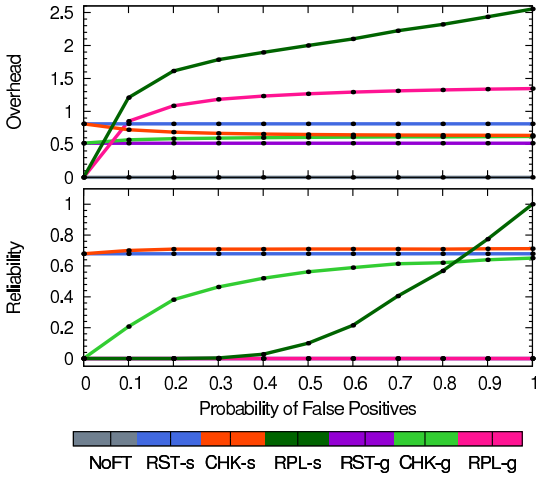
i. The execution of a proactive strategy increases with an increase in the probability of false positives. Though the rise in an unnecessary proactive strategy execution is generally considered as a waste of resources, there are circumstances when such event pays off. Suppose a proactive strategy is executed to manage the failure of an activity due to a false positive prediction. If this activity fails later in the execution *and* the impending



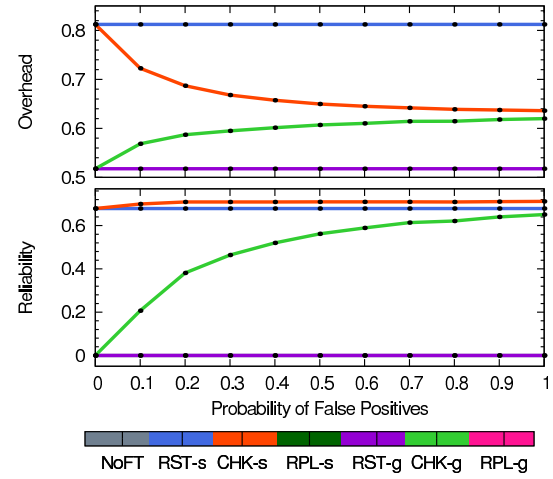
(a) Probability of activity failure 0.01.



(b) Probability of activity failure 0.01. Magnified CHK-s, CHK-g, RST-s, RST-g.



(c) Probability of activity failure 0.195.



(d) Probability of activity failure 0.195. Magnified CHK-s, CHK-g, RST-s, RST-g.

Figure 9.6: The impact of the accuracy of a state oblivious predictor on the reliability-overhead tradeoff: the impact of an increase in the probability of false positives is read from left to right, and the impact of an increase in the probability of false negatives is read from right to left.

failure of the activity is not identified at the most recent prediction, then the previously executed proactive strategy presents an opportunity to successfully complete the failed activity.

ii. In a state oblivious predictor, if the probability of false positives increases, so does the probability of true positives (Section 8.2.1.2). Increasing the probability of true positives is a desirable property since it presents an opportunity to take an action that potentially minimises the impact of the impending failure on the overall computation.

Overall, increasing the probability of false positives progressively increases the use of a proactive strategy and decreases the use a reactive strategy. This property is advantageous for both replication-based and checkpointing-based RAC:

- The replication-based RAC manages failure solely proactively. Since increasing the probability of false positives decreases the reliance on a reactive strategy to manage failure, the replication-based RAC gains a performance boost as the probability of false positives gets higher and higher.

- In the checkpointing-based RAC, increasing the probability of false positives results in more activities being checkpointed than before. If/when activities fail, provided that the failure is recoverable, they will be rolled-back to the last checkpoint. Since rolled-back activities generally have shorter execution time than restarted ones, they have a better chance of completing successfully.

The extent of the performance gain due to increased positive predictions depends on the type of the fault tolerance support and the behaviour of the computation. Given failure is not impending, if the probability of false positives is 1, the cause of an activity failure is a transient fault and an activity does not fail before the first prediction is made, then the MR-specific replication-based RAC guarantees 100% reliable computation. In this scenario, the predictor always sends warning, and subsequently all activities are protected at all times irrespective of their status of computation. However, this is not the case for the generic replication-based RAC. Even though the predictor sends warning all the time, the generic replication-based RAC does not know how to replicate an activity (e.g., reduce) that depends on a previously completed activity (e.g., map). Once the point, where the failure of all independent activities can be handled, is reached, increasing the probability of false positives will not yield any significant reliability gain. Figure 9.6a shows that, for example, the generic replication-based

RAC increases the reliability of an MR application execution considerably until the probability of false positives reaches 0.7. Once such point is reached, reliability remains almost constant.

Similar to the replication-based RAC, for the checkpointing-based RAC, with increasing probability of false positives, reliability increases. However, as Figure 9.6 shows, reliability approaches a limit asymptotically. This limit seems to represent a residual amount of unrecoverable failure, which cannot be managed by checkpointing. For example, Figure 9.6c shows that the generic checkpointing-based RAC asymptotically approaches 0.68.

Despite an increase in the probability of false positives having a desirable impact on reliability, it increases the overhead of all, except the MR-specific checkpointing-based RAC. Though the execution of a proactive strategy due to a false positive prediction is acceptable in some circumstances, it wastes resources in large part. This is why, in general, an increase in the probability of false positives leads to increased overhead. The extent of the overhead increment, however, is bounded. In the replication-based RAC, as discussed in Section 7.2.4, there are at most two copies of a given activity that are running at the same time. Thus, no matter how many times a predictor sends a warning, which will be frequent when the probability of false positives is high, an activity will not be replicated as long as two copies of the activity are being executed.

In the checkpointing-based RAC, with an increase in the probability of false positives, the overhead of the generic fault tolerance support increases, but the overhead of the MR-specific one decreases. When the probability of false positives increases, so does the probability for the activities to be checkpointed before their impending failure. Since the generic checkpointing-based RAC can manage the failure of checkpointed activities, including the ones whose computation depend on others, the more activities are checkpointed before their failure, the more the overhead of the generic checkpointing-based RAC will be. In the MR-specific checkpointing-based RAC, activity failure is managed by rollback if the activity is checkpoined, and by restart if the activity is not checkpointed. The more activities are checkpointed, the more the failure of activities is managed by rollback. Therefore, under the default parameter settings, since rollback is cheaper than restart, the overhead of the MR-specific checkpointing-based RAC decreases. If the cost of a single checkpoint is very expensive, of course, the total time that is saved by rolling-back instead of restarting a failed activity could be eclipsed by the total checkpointing overhead.

9.5.2 State Aware Predictors

We discuss the impact of the accuracy of a state aware predictor on the reliability-overhead tradeoff. In a state aware predictor, the probabilities of false positives and false negatives do not have the same relationship as they do in a state oblivious predictor. Therefore, we separately examine how each variable affects reliability and overhead. Finally, we highlight what the reliability of a grid computation looks like if the replication-based and the checkpointing-based RAC are fitted with the perfect predictor. In summary:

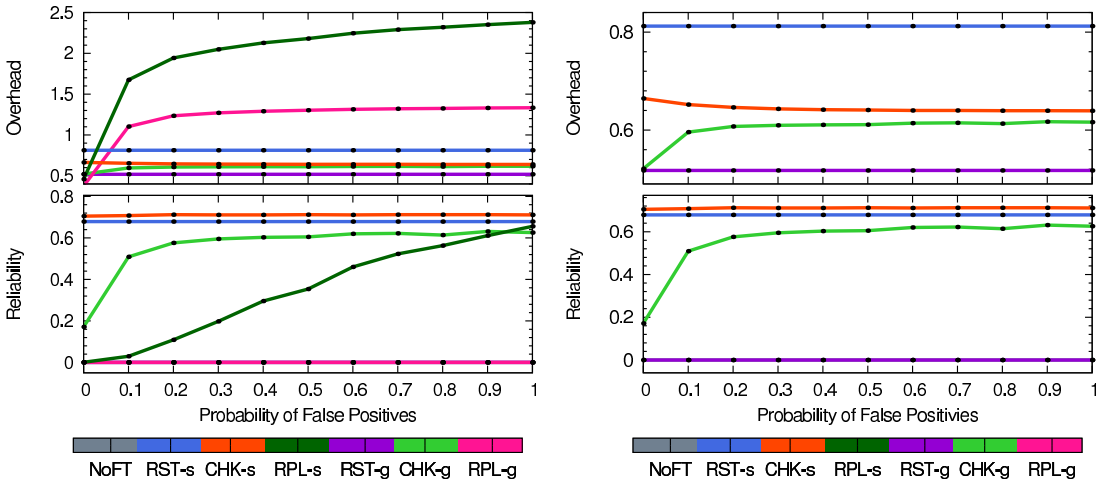
- *Result 24*: As the probability of false positives in a state aware predictor increases, the replication-based and the checkpointing-based RAC progressively increase the reliability of an MR application up to a limit. Nonetheless, except for the MR-specific checkpointing-based RAC, they incur more and more overhead. Figure 9.7.

- *Result 25*: As the probability of false negatives in a state aware predictor increases, the replication-based and the checkpointing-based RAC provide less and less reliable execution of MR applications. However, except for the MR-specific checkpointing-based RAC, they require less and less overhead. Figure 9.8.

- *Result 26*: With the perfect predictor, under the default parameter settings, the MR-specific replication-based RAC achieves the highest reliable computation, followed by the MR-specific checkpointing-based, the generic checkpointing-based, and finally the generic replication-based RAC. Figure 9.9.

9.5.2.1 False Positives

Suppose n be the probability of false positives given failure is not impending, and m be the probability of false negatives given failure is impending. In this evaluation, $n \in [0, 1]$ and $m = 0.2$. The impact of the probability of false positives of a state aware predictor is similar to that of a state oblivious predictor. As shown in Figure 9.7, as the probability of false positives in a state aware predictor increases, the replication-based and the checkpointing-based RAC increase the reliability of an MR grid application up to a limit. However, these fault tolerance support types, except for the MR-specific checkpointing-based RAC, require higher and higher overhead. See Section 9.5.1 for further discussion.



(a) Probability of activity failure 0.195.

(b) Probability of activity failure 0.195. Magnified CHK-s, CHK-g, RST-s, RST-g.

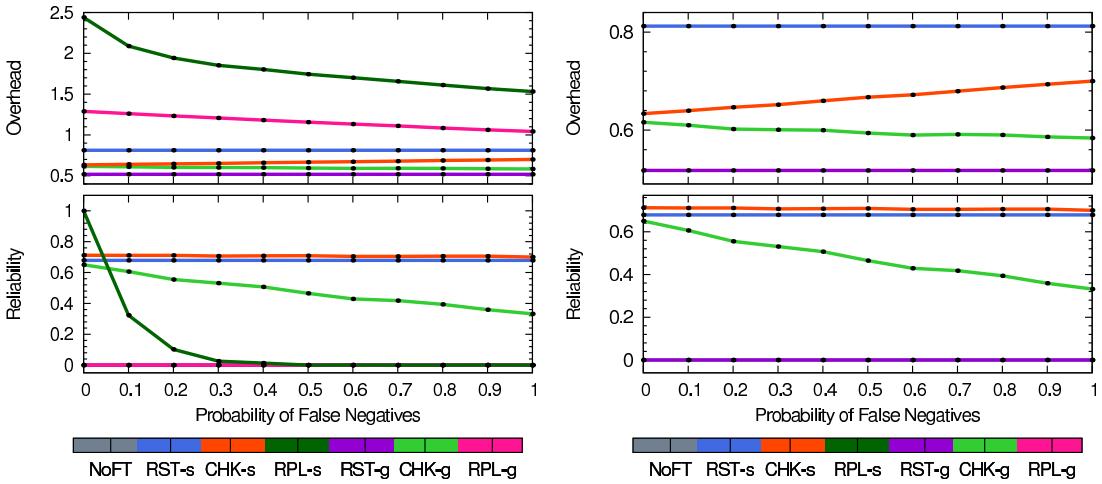
Figure 9.7: The impact of the probability of false positives of a state aware predictor on the reliability-overhead tradeoff.

9.5.2.2 False Negatives

Suppose n be the probability of false positives given failure is not impending, and m be the probability of false negatives given failure is impending. In this evaluation, $m \in [0, 1]$ and $n = 0.2$.

As the probability of false negatives increases, as shown in Figure 9.8, the checkpointing-based and the replication-based RAC provide less and less reliable execution of an MR application. With an increase in the probability of false negatives, the number of impending failures that will be predicted decreases. As the result, the number of activities whose failures should be handled by a reactive strategy increases. If the given fault tolerance support can manage the failure of such activities, like the MR-specific checkpointing-based RAC, then its performance with respect to reliability will not be significantly affected. Otherwise, the reliability of the computation will significantly decrease. Due to the absence of a reactive a strategy in the replication-based RAC, and a mechanism to recover an uncheckpointed successor activity by the generic checkpointing-based RAC, as shown in Figure 9.8, the performance of these fault tolerance support types with respect to reliabilty deteriorates as the probability of false negatives increases.

As shown in Figure 9.8, an increase in the probability of false negatives decreases the overhead of the replication-based and the generic checkpointing-based RAC, but increases the



(a) Probability of activity failure 0.195.

(b) Probability of activity failure 0.195. Magnified CHK-s, CHK-g, RST-s, RST-g.

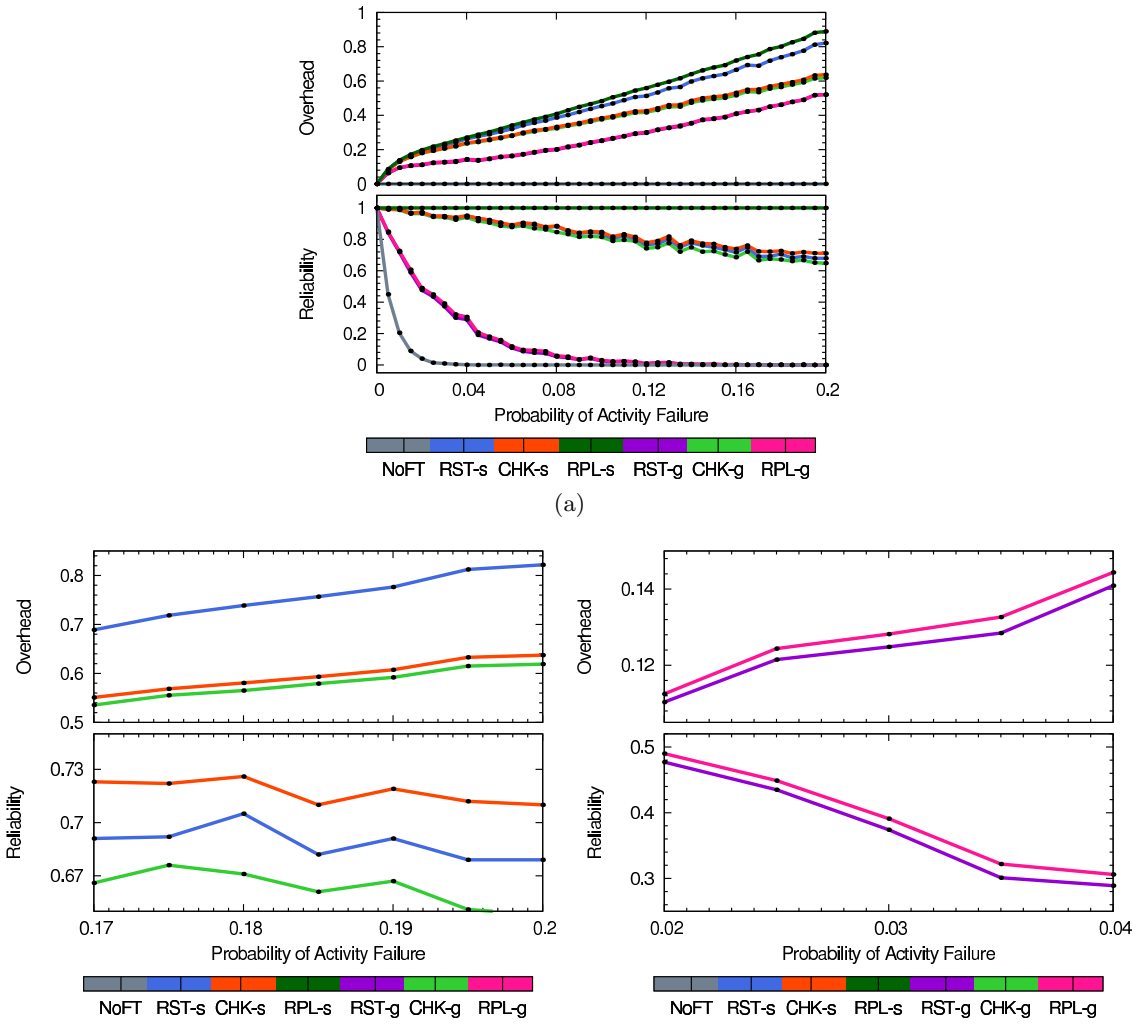
Figure 9.8: The impact of the probability of false negatives of a state aware predictor on the reliability-overhead tradeoff.

overhead of the MR-specific checkpointing-based RAC. The decrease in overhead occurs due to the fault tolerance support types not putting any effort to recover the failed activities whose impending failure was not predicted. The increase in the overhead of the MR-specific checkpointing-based RAC is the result of reactive handling of the failure of activities whose impending failures are not predicted. The extent of the overhead increase depends on the number of the failed activities. If significantly many of the activities of the application fail, the overhead will be significantly high. For instance, as shown in Figure 9.8b, an increase in probability of false negatives incurs up to 10% more overhead than before.

9.5.2.3 The Perfect Predictor

In an ideal prediction world, where there are no false positives and false negatives, under the default parameter settings, the MR-specific replication-based RAC achieves the highest reliable computation, followed by the MR-specific checkpointing-based, the MR-specific restart-based, the generic checkpointing-based, and the generic replication-based RAC. The least reliable computation is provided by the generic restart-based RAC.

As discussed in Section 9.1.1.4, due to the non-zero probability of unrecoverable failure, the MR-variants of the restart-based and the checkpointing-based RAC provide less reliable computation than their replication counterpart. If the probability of unrecoverable failure is



(a)

(b) Magnified CHK-s, RST-s and CHK-g

(c) Magnified RPL-g and RST-g

Figure 9.9: The reliability-overhead tradeoff with the perfect predictor.

zero, as discussed in Section 9.2, they do provide 100% reliable computation.

The MR-specific checkpointing-based RAC provides marginally higher reliability than its generic counterpart, as shown in Figure 9.9b. Despite the predictor being perfect, some successor activities may fail before the first prediction is made. This makes the generic checkpointing-based RAC not provide equally reliable computation as its MR-specific counterpart.

The generic replication-based RAC has absolutely no mechanism to handle the failure of successor activities. As the result, even though the predictor sends a warning about the impending failure of a successor activity, no action will be taken. As for the generic restart-based RAC, since such fault tolerance support does not assume prediction, the accuracy of

the predictor has no influence on its performance. This, of course, is true for the MR-specific restart-based RAC as well.

9.6 Reflection

We demonstrated in Sections 9.1-9.5 that there is no single best reliability-overhead tradeoff that characterises the performance of a given RAC architecture in all scenarios. Our data shows that the choice of a good tradeoff depends on the classification of the architecture of the given grid application, the type of the fault tolerance strategy with which the RAC architecture is paired, the values of the parameters of the fault tolerance strategy and/or the accuracy and the interval of predictions. A small change in the value of a parameter may significantly affect the reliability-overhead tradeoff, for example, consider the impact of a change in the probability of false negatives from 0 to 0.1 on the replication-based RAC (Figure 9.8a). On another scenario, a change in the parameter value may not have a notable impact on the reliability-overhead tradeoff, for example, consider the impact of using checkpointing mechanisms whose overheads are 0.01% and 1% of activity execution time on the checkpointing-based RAC (Figure 9.4d).

Even for a given architecture type, there may not be a single best tradeoff of reliability and overhead. The best tradeoff may depend on user requirements, or the cost of resources available to the user. Hence, the results and methods presented in this thesis may be useful in making such tradeoffs. Suppose an application programmer develops an MR application that resembles the scenario depicted in Figure 9.1b, the probability of activity failure is roughly 0.12, and the parameter settings of the execution environment resembles the default parameter settings of our experiment design. As shown in the figure, six fault tolerance support types are available. Since all of the generic ones do not improve reliability, they will not be further investigated. Now, the choice of three MR-specific RAC rests on the requirements of the application programmer. If the requirement is to achieve the highest reliability, then the choice is clearly the replication-based RAC. The replication-based RAC guarantees almost 100% reliability, but increases the cost of the computation by 100%. If the requirement is to achieve at least 80% reliability with the least overhead possible, then the choice is the checkpointing-based RAC. Both the checkpointing-based and the restart-based RAC guarantee the required reliability; but restart incurs 10% overhead while checkpointing incurs only 1% overhead.

Even though checkpointing incurs the smallest computational overhead, adapting checkpointing requires more effort than restart. In this thesis, we did not evaluate the developmental cost of a RAC-based grid system. However, it is reasonable to expect that the developmental

cost of the restart-based RAC will be cheaper than the checkpointing one. Developing checkpointing requires identifying what to checkpoint, the type of checkpointing, the location for saving the checkpointed data, how to access the data during rollback, and how to rollback. In contrast, restart only needs access to the input data of the computation and the executable file of the computation. Now, the question is *is it worth to go through all the trouble of developing checkpointing-based RAC to save 9% computational overhead?* If not, then the best choice that satisfies the requirement becomes the restart-based RAC.

The result of our experiments offers insight into the diversity of fault tolerance support configurations and their respective reliability-overhead tradeoff. The results support grid administrators, service managers, and/or grid software developers in making decisions to improve reliability or to decrease overhead of reliability improvement or both.

9.7 Summary

In this chapter, we evaluated the reliability-overhead tradeoffs that are enabled by the generic, the MR-specific and the CL-specific RAC architectures. We paired these architectures with restart, replication and checkpointing-rollback fault tolerance strategies. We have shown the tradeoff of each architecture under the default experimental settings, and then explored the impact of unrecoverable failure, the cost of a single checkpoint, and the accuracy and the interval of a predictor on the tradeoff. We finally reflected on what we have learnt at the end of this project.

Chapter 10

Conclusion

"Now this is not the end.
It is not even the beginning of the end.
But it is, perhaps, the end of the beginning."
- Winston Churchill

This PhD project contributes the RAC approach, which is a fault tolerance approach that manages failure at the component level, combines reactive and proactive fault tolerance strategies, assumes runtime prediction with proactive failure management, and provides customized fault tolerance support based on the classification of the architecture of a grid application. Further, the project provides parameterised Markov models and testbed for reliability and overhead analyses. We have used the testbed for evaluating the reliability-overhead tradeoff of the RAC approach. Via simulated experiment, we have confirmed that the architecture-specific fault tolerance support provides higher reliability improvement and incurs higher overhead to grid applications than the architecture-unaware one. The degree of the reliability improvement of the architecture-specific support over the architecture-unaware one depends on factors like the type of the fault tolerance strategy selected and its parameters, and the accuracy of a predictor. In this chapter, we summarize our contributions, the key research results, and finally future work.

10.1 Contributions

This research project contributes the following [Yusuf et al., 2009, 2011, Yusuf, 2010]:

i. The RAC approach, a novel prediction and architecture-based fault tolerance approach that enables grid applications to tolerate failure reactively and proactively at the component level.

ii. A new formal Markov-chain based model that marries assumptions of the so-called Bulk-Synchronous Parallel model with specific classes of application architectures known as dwarfs. Specifically, the MapReduce and Combinational Logic dwarfs are modelled formally in this framework for the first time.

iii. Thorough and detailed reliability-overhead analyses of the RAC approach under varying assumptions and conditions.

iv. In-depth analyses of the impact of unrecoverable failure, checkpointing cost, prediction interval and predictor accuracy on the reliability-overhead tradeoff of the RAC approach.

v. A parameterised experiment testbed that enables a grid fault tolerance expert to evaluate diverse fault tolerance support configurations, and then choose the one that will satisfy the reliability and cost requirements.

10.2 Key Results

This thesis offers insight into providing customized, but not application-specific, fault tolerance support to a wide range of grid applications. By considering a small set of parameters that are shared among many grid applications, the thesis demonstrated that fine-grained and flexible fault tolerance support leads to significant reliability improvement. We also showed the increased overheads associated with such reliability improvement.

The thesis explored the nature of the reliability-overhead tradeoff for grid applications that are classified under the MapReduce and the Combinational Logic dwarfs. We learnt that the exact reliability-overhead tradeoff depends on many factors. However, one can use either the RACS models or the experiment testbed to find out which combination of parameter values leads to the desired level of reliability improvement, and the overhead of the improvement on the overall computation. The key factors that determine the nature of the reliability-overhead tradeoff, and their impact on the tradeoff, are as follows:

- *Type of RAC architecture*: The architecture-specific fault tolerance support provides a higher reliability improvement to MapReduce (MR) and Combinational Logic (CL) grid applications than the generic one, all else being equal (Results 1 and 8). The architecture-specific fault tolerance support incurs higher overhead to the overall computation than the generic fault tolerance support, except when the execution time of the map activity is significantly higher than the execution time of the reduce activity (in MR only) and when failure is managed by checkpointing-rollback (Results 2, 3 and 9). In these situations, the overhead of the architecture-specific and the generic fault tolerance support is similar.

- *Type of fault tolerance strategy*: Of the architecture-specific fault tolerance support types, checkpointing and restart provide equally reliable grid computation (Results 4 and 11). Furthermore, given that the failure of some activities requires reactive fault tolerance management, these fault tolerance support types provide more reliable grid application execution than the replication-based architecture-specific fault tolerance support (Results 5 and 12). Given that a proactive strategy can be executed prior to the failure of any activity and the cause of the failure is a transient fault, the replication-based architecture-specific fault tolerance support guarantees 100% reliable execution (Results 6 and 13). Given that the probability of activity failure is non-zero, replication introduces the highest overhead among the architecture-specific fault tolerance support types, followed by restart and then checkpointing (Results 7 and 14).

- *Probability of unrecoverable failure*: As the probability of unrecoverable failure increases, the reliability improvement that is provided by the restart-based and the checkpointing-based fault tolerance support types decreases, all else being equal (Result 15). Likewise for overhead (Result 16). When the cause of the unrecoverable failure is not an internal software fault, an increase in the probability of unrecoverable failure does not have a notable impact on the replication-based fault tolerance support, all else being equal (Result 17).

- *Cost of single checkpoint*: As the cost of a single checkpoint increases, the reliability of the computation whose failure is managed by the checkpointing-based fault tolerance support either stays the same or decreases, provided that the selected prediction interval enables the checkpointing-based support to provide the most reliable computation with the least overhead possible (Result 18). Given that reliability remains almost constant,

an increase in the cost of a single checkpoint generally increases the overhead of the checkpointing-based fault tolerance support (Result 19).

- *Prediction interval*: With an increase in the prediction interval, the reliability of a computation whose failure is managed by the replication-based fault tolerance support decreases (Result 20). An increase in the prediction interval decreases the degree of proactive fault tolerance support provision, and increases the reliance on reactive strategies for failure management. Since replication-based fault tolerance support manages failure only proactively, the increasing reliance on a reactive strategy to manage failure limits its performance. With an increase in the prediction interval, the reliability of a computation whose failure is managed by the checkpointing-based fault tolerance support increases (Result 21). The increase in the reliability continues until the prediction interval reaches its optimal value. The optimal prediction interval enables the checkpointing-based fault tolerance support to provide the highest possible reliability improvement to a grid application, under the default parameter settings. If the prediction interval is larger than the optimal value, the reliability of the computation remains constant in the architecture-specific fault tolerance support, but decreases in the generic fault tolerance support.

- *Prediction accuracy*: As the probability of false positives increases, the replication-based and the checkpointing-based fault tolerance support progressively increase the reliability of a grid application up to a limit, all else being equal (Results 22 and 24). As the probability of false negatives increases, the restart-based and the checkpointing-based fault tolerance support provide less reliable execution of grid applications than before (Result 25). Given a non-zero probability of unrecoverable failure, the cause of failure is a transient fault, a relatively cheap cost of checkpointing, optimal prediction interval and the perfect predictor, the replication-based architecture-specific fault tolerance support achieves the highest reliable computation, followed by checkpointing-based architecture-specific, restart-based architecture-specific, checkpointing-based generic, and finally restart-based generic fault tolerance support (Result 26). With an increase in the probability of positives, the replication-based and the generic checkpointing-based fault tolerance support incur more and more overhead (Results 23 and 24); whereas an increase in the probability of false negatives leads to less and less overhead from these fault tolerance support types (Result 25).

10.3 Future Work

In the future, it is interesting to explore whether the RAC approach enables similar reliability-overhead tradeoff in grid applications other than MapReduce and Combinational Logic, and to confirm the exact nature of reliability-overhead tradeoffs for the other typical dwarfs underlying grid applications. Clearly, not all dwarfs are well suited to be executed on a grid infrastructure. For example, the Finite State Machine dwarf represents applications that are embarrassingly sequential. Thus, perhaps, a grid is not needed for such a computation. However, it is still worthwhile to study how one might leverage the knowledge about the dwarfs to guarantee highly reliable computations irrespective of the computing environment.

አለቀ።

Bibliography

http://hadoop.apache.org/. Cited on page 85.

Standard Performance Evaluation Corporation. URL http://www.spec.org/index.html. Last visited on February 14^{th}, 2012. Cited on page 42.

http://www.macresearch.org/the_xgrid_tutorials. Cited on page 85.

Pallas MPI Benchmarks - PMB, Part MPI-1. Technical Report, 2000. Cited on page 120.

The Early Detection Research Network: Investing in Translational Research on Biomarkers of Early Cancer and Cancer Risk. Technical Report 4, National Cancer Institute, January 2008. URL http://prevention.cancer.gov/files/edrn4th.pdf. Cited on page 17.

D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A Tool for Performing Parameterized Simulations Using Distributed. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*, HPDC '95, pages 112–, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7088-6. URL http://dl.acm.org/citation.cfm?id=822081.823037. Cited on page 16.

D. Abramson, J. Giddy, and L. Kotler. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? In *Proceedings of the 14th International Parallel and Distributed Processing Symposium, (IPDPS 2000)*, pages 520 –528, 2000. Cited on page 16.

Amazon Website. Amazon EC2 Pricing, 2012. URL http://aws.amazon.com/ec2/pricing/. Last visited on March 14^{th}, 2012. Cited on page 118.

K. Amin, G. V. Laszewski, M. Hategan, N. J. Z. S. Hampton, and A. Rossi. GridAnt: A Client-Controllable Grid Workflow System. In *Proceedings of the 37th Hawaii International Conference on System Science*, pages 1–10, January 2004. Cited on page 21.

Mac OS X Server Xgrid Administration and High Performance Computing Version 10.6 Snow Leopard. Apple Inc., 2009. Cited on page 16.

M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Communication of ACM*, 53:50–58, 2010. Cited on page 22.

K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, USA, December 2006. Cited on pages 3, 42, 43, 82, 91, and 92.

K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. D. Kubiatowicz, E. A. Lee, N. Morgan, G. Necula, D. A. Patterson, K. Sen, J. Wawrzynek, and K. A. Wessel, David Yelick. The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View. Technical Report UCB/EECS-2008-23, University of California, Berkeley, USA, March 2008. Cited on pages 3 and 42.

K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A View of the Parallel Computing Landscape. *Communications of the ACM*, 52:56–67, 2009. ISSN 0001-0782. Cited on pages 3 and 43.

Australian Synchrotron, January 2012. URL http://www.synchrotron.org.au/. Cited on page 10.

A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, 11:1491–1501, December 1985. ISSN 0098-5589. Cited on page 29.

A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11 – 33, 2004. Cited on pages 24 and 25.

B. Bode, D. M. Halstead, R. Kendall, Z. Lei, and D. Jackson. The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase & Conference*, Berkeley, CA, USA, 2000. USENIX Association. Cited on page 13.

G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley Object Technology Series. Addison-Wesley Professional, 2nd edition, 2005. ISBN 0321267974. Cited on pages 30 and 59.

I. Brandic and S. Dustdar. Grid vs Cloud A Technology Comparison. *it - Information Technology*, 53(4):173–179, 2011. Cited on page 22.

F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner. Architecture-Based Reliability Prediction with the Palladio Component Model. *IEEE Transactions on Software Engineering*, PP (99):1–20, September 2011. ISSN 0098-5589. Cited on pages 30 and 40.

D. Bucciarelli. Java Grid Computing, 2012. URL http://jcgrid.sourceforge.net/. Last visited on January 17^{th}, 2012. Cited on page 16.

K. Budati, J. Sonnek, A. Chandra, and J. Weissman. RIDGE: Combining Reliability and Performance in Open Grid Platforms. In *Proceedings of the 16th international symposium on High performance distributed computing*, HPDC '07, pages 55–64, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-673-8. Cited on pages 2, 3, 48, 49, 50, 53, and 54.

R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. In *Proceedings of the 4th International Conference on High Performance Computing in Asia-Pacific Region (HPC ASIA 2000)*, pages 283–289. IEEE Computer Society Press, 2000. Cited on page 16.

C. Campbell. Xgrid Agent for Java, August 2005. URL http://sourceforge.net/projects/xgridagent-java/. Cited on page 16.

E. Cesario and D. Talia. A Failure Handling Framework for Distributed Data Mining Services on the Grid. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 70 –79, February 2011. Cited on page 50.

W. Chen, L. Zong, W. Huang, G. Ou, Y. Wang, and D. Yang. An Empirical Study of Massively Parallel Bayesian Networks Learning for Sentiment Extraction from Unstructured Text. *Lecture Notes in Computer Science*, 6612:424–435, 2011. Cited on page 84.

Y. Chen, D. Pavlov, and J. F. Canny. Large-Scale Behavioral Targeting. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'10)*, pages 209–217. ACM, 2009. Cited on page 84.

L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik. Early Prediction of Software Component Reliability. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 111–120, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. Cited on page 30.

R. C. Cheung. A User-Oriented Software Reliability Model. *IEEE Transactions on Software Engineering*, SE-6(2):118 – 125, March 1980. ISSN 0098-5589. Cited on pages 30 and 36.

M. Chtepen, F. H. A. Claeys, B. Dhoedt, F. De Turck, P. Demeester, and P. A. Vanrolleghem. Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids. *IEEE Transactions Parallel Distributed Systems*, 20:180–190, February 2009. ISSN 1045-9219. Cited on pages 2, 48, 49, 53, and 54.

P. Colella. Defining Software Requirements for Scientific Computing. Presentation, 2004. Cited on page 42.

V. Cortellessa, H. Singh, and B. Cukic. Early reliability assessment of uml based software models. In *Proceedings of the 3rd international workshop on Software and performance*, WOSP '02, pages 302–309, New York, NY, USA, 2002. ACM. ISBN 1-58113-563-7. Cited on page 30.

D. Cote. XGrid agent for Unix architectures, June 2004. URL http://www.novajo.ca/xgridagent/. Cited on page 16.

P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger. Workflow Management in Condor. In I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 357–375. Springer London, 2007. Cited on pages 20, 21, and 51.

D. Crawl and I. Altintas. A Provenance-Based Fault Tolerance Mechanism for Scientific Workflows. In J. Freire, D. Koop, and L. Moreau, editors, *Provenance and Annotation of Data and Processes*, volume 5272 of *Lecture Notes in Computer Science*, pages 152–159. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-89964-8. Cited on page 52.

D. Crisp et al. The Orbiting Carbon Observatory (OCO) mission, 2003. Cited on page 17.

DAS4 Website. The Distributed ASCI Supercomputer 4, January 2012. URL http://www.cs.vu.nl/das4/. Cited on page 10.

DataGrid Website. The EU DataGrid Project, January 2012. URL http://www.eu-datagrid.org/. Cited on page 10.

DataTAG Website. Research & technological development for a **Data** **T**rans**A**tlantic **G**rid, January 2012. URL http://datatag.web.cern.ch/datatag/. Cited on page 10.

J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI 04: 6th Symposium on Operating System Design and Implementation*, December 2004. Cited on pages 2, 4, 48, 50, 53, and 84.

J. Dean et al. MapReduce: Simplified Data Processing on Large Clusters. *ACM Comm.*, 51 (1):107–113, 2008. Cited on page 84.

E. Deelman, G. Singh, M. hui Su, J. Blythe, A. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13:219–237, 2005. Cited on page 21.

R. P. Dick, D. L. Rhodes, and W. Wolf. Tgff: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, CODES/CASHE '98, pages 97–101, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8442-9. Cited on page 108.

The DSpace Developer Team, 2011. *DSpace 1.8 Documentation*. The DSpace Foundation, November 2011. Cited on page 16.

R. Duan, R. Prodan, and T. Fahringer. DEE: A Distributed Fault Tolerant Workflow Enactment Engine for Grid Computing. In L. Yang, O. Rana, B. Di Martino, and J. Dongarra, editors, *High Performance Computing and Communications*, volume 3726 of *Lecture Notes in Computer Science*, pages 704–716. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-29031-5. Cited on pages 3, 52, and 53.

Dwarf Mine Website. Dwarf Mine, 2012. URL http://view.eecs.berkeley.edu/wiki/Dwarf_Mine. Last visited on February 14^{th}, 2012. Cited on pages 42 and 43.

EC2. Amazon Elastic Compute Cloud (Amazon EC2), 2012. URL http://aws.amazon.com/ec2/. Last visited on January 24th, 2012. Cited on page 23.

T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto, Jr., and H.-L. Truong. ASKALON: A Tool Set for Cluster and Grid Computing. *Concurr. Comput. : Pract. Exper.*, 17:143–169, February 2005. ISSN 1532-0626. Cited on page 21.

A. Field and P. Harrison. *Functional Programming*. Addison-Wesley, Wokingham, England, 1988. Cited on page 82.

M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans on Computers*, C-21:948–960, 1972. Cited on pages 43 and 91.

I. Foster. What is the Grid? A Three Point Checklist. *GRIDtoday*, 1(6), 2002. URL http://dlib.cs.odu.edu/WhatIsTheGrid.pdf. Cited on pages 10 and 19.

I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. *Journal of Computer Science and Technology*, 21(4):513–520, July 2006. Cited on pages 16 and 17.

I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999. ISBN 1-55860-475-8. Cited on page 1.

I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *Proceedings of the International Workshop on Quality of Service*, pages 27–36, 1999. Cited on page 13.

I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications.*, 15: 200–222, August 2001. ISSN 1094-3420. Cited on pages 1, 10, 11, 12, 13, and 14.

I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1 –10, November 2008. Cited on page 22.

I. Foster et al. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Globus Project, June 2002. URL http://www.globus.org/alliance/publications/papers/ogsa.pdf. Cited on page 14.

I. Foster et al. The Open Grid Services Architecture, Version 1.0. Global Grid Forum, January 2005. URL http://www.gridforum.org/documents/GWD-I-E/GFD-I.030.pdf. Cited on pages 14 and 15.

A. Fox and D. Patterson. Guest Editors' Introduction: Approaches to Recovery-Oriented Computing. *IEEE Internet Computing*, 9(2):14 – 16, March-April 2005. ISSN 1089-7801. Cited on page 38.

G. C. Fox and D. Gannon. Workflow in Grid Systems: Editorials. *Concurr. Comput. : Pract. Exper.*, 18:1009–1019, August 2006. Cited on pages 19, 20, and 21.

J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5:237–246, July 2002. Cited on pages 2 and 16.

D. Garlan and M. Shaw. An Introduction to Software Architecture. Technical report, Pittsburgh, PA, USA, 1994. URL http://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf. Cited on pages 40, 41, and 42.

L. Garotte and R. L. Bras. A Distributed Model for Real-Time Flood Forecasting using Digital Elevation Models. *Journal of Hydrology*, 167:279–306, 1995. Cited on page 92.

Globus Website, 2012. URL http://www.globus.org/. Last visited on January 19^{th}, 2012. Cited on page 17.

P. K. GM, K. P. Leela, M. Parsana, and S. Garg. Learning Website Hierarchies for Keyword Enrichment in Contextual Advertising. In *Proceedings of the ACM International Conference on Web Search and Data Mining (WSDM'11)*, pages 425–434. ACM, February 2011. Cited on page 84.

GoGrid Website. GoGrid Pricing, 2012. URL http://www.gogrid.com/cloud-hosting/cloud-hosting-pricing.v2.php. Last visited on March 14^{th}, 2012. Cited on page 118.

S. Gokhale, W. Wong, K. Trivedi, and J. Horgan. An Analytical Approach to Architecture-Based Software Performance and Reliability Prediction. In *Proceedings. IEEE International Computer Performance and Dependability Symposium*, pages 13 –22, sep 1998. doi: 10.1109/IPDS.1998.707705. Cited on page 40.

Google App Engine. Google App Engine, 2012. URL http://code.google.com/appengine/. Last visited on January 24th, 2012. Cited on page 23.

M. I. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. PhD thesis, Massachusetts Institute of Technology, June 2010. Cited on page 121.

R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *Computer*, 30: 68–74, April 1997. Cited on pages 28 and 29.

D. Hollinsworth. The Workflow Reference Model. Tc00-1003, Hampshire, UK, January 1994. URL http://www.wfmc.org/standards/docs/tc003v11.pdf. Cited on page 19.

Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 381 –390, June 1995. doi: 10.1109/FTCS.1995.466961. Cited on pages 28 and 29.

S. Hwang and C. Kesselman. GridWorkflow: A Flexible Failure Handling Framework for the Grid. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, HPDC '03, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1965-2. Cited on pages 2, 51, and 53.

IEEE. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, 1990. Cited on pages 24, 25, 26, 30, and 38.

IEEE. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. *IEEE Std 1471-2000*, 2000. doi: 10.1109/IEEESTD.2000.91944. Cited on page 40.

iRODS Website. iRODS: Integrated Rule-based Data System, 2012. URL http://www.irods.org/. Last visited on January 17th, 2012. Cited on page 16.

P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, Englewood Cliff, New Jersey, 1994. Cited on pages 26, 27, 28, 29, and 38.

Planetary Data System Archive Preparation Guide (APG) . Jet Propulsion Laboratory, April 2010. Cited on page 17.

H. Jurgen and F. Thomas. Synthesizing Byzantine Fault-Tolerant Grid Application Wrapper Services. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 467–474, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3156-4. Cited on page 50.

G. Kandaswamy, A. Mandal, and D. A. Reed. Fault Tolerance and Recovery of Scientific Workflows on Computational Grids. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 777–782, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3156-4. Cited on pages 2 and 51.

M. Karimzadehgan, W. Li, R. Zhang, and J. Mao. A Stochastic Learning-To-Rank Algorithm and its Application to Contextual Advertising. In *Proceedings of the 20th International Conference on World Wide Web (WWW'11)*, pages 377–386. ACM, March 2011. Cited on page 84.

R. Kuntschke, T. Scholl, S. Huber, A. Kemper, A. Reiser, H.-M. Adorf, G. Lemson, and W. Voges. Grid-Based Data Stream Processing in e-Science. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, E-SCIENCE '06, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2734-5. Cited on pages 4, 92, and 121.

M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011. Cited on pages 5 and 30.

Y. Li and M. Mascagni. Improving Performance via Computational Replication on a Large-Scale Computational Grid. In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, CCGRID '03, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1919-9. Cited on pages 49, 50, and 54.

M. Litzkow et al. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report, University of Wisconsin-Madison, April 1997. Cited on pages 16 and 28.

D. Logofatu and D. Dumitrescu. Parallel Evolutionary Approach of Compaction Problem Using MapReduce. *Lecture Notes in Computer Science*, 6239:361–370, 2011. Cited on page 84.

B. Ludscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006. Cited on page 21.

A. Luther, R. Buyya, R. Ranjan, and S. Venugopal. Alchemi: A .NET-based Enterprise Grid Computing System. In *Proceedings of the 6th International Conference on Internet Computing (ICOMP'05)*, 2005. Cited on page 16.

MathWorks Website. MathWorks Website. URL http://www.mathworks.com.au/. Last visited on February 6^{th}, 2012. Cited on pages 5, 30, 67, 73, 76, 78, 90, and 102.

C. A. Mattmann, S. Malek, N. Beckman, M. Mikic-rakic, N. Medvidovic, and D. J. Crichton. GLIDE: A Grid-based Lightweight Infrastructure for Data-intensive Environments. In *European Grid Conference*, 2005. Cited on page 17.

P. B. Moranda. Predictions of Software Reliability during Debugging. In *Proceedings of the Annual Reliability and Maintainability Symposium*, 1975. Cited on page 30.

J. Musa. *Software Reliability Engineering*. McGraw-Hill, 1998. Cited on page 24.

J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, Professional edition, 1990. Cited on page 24.

myGrid Website, 2012. URL http://www.mygrid.org.uk/. Last visited on January 26^{th}, 2012. Cited on page 22.

B. Nazir, K. Qureshi, and P. Manuel. Adaptive Checkpointing Strategy to Tolerate Faults in Economy based Grid. *Journal of Supercomputing*, 50:1–18, October 2009. ISSN 0920-8542. Cited on pages 2, 48, 49, and 53.

NIST. Data Encryption Standard. *Federal Information Processing Standards*, (46-3), October 1999. Cited on pages 4 and 92.

NIST. Advanced Encryption Standard. *Federal Information Processing Standards*, (197), November 2001. Cited on pages 4 and 92.

D. Nurmi, J. Brevik, and R. Wolski. Minimizing the network overhead of checkpointing in cycle-harvesting cluster environments. In *Proceedings of IEEE International Cluster Computing*, pages 1 –10, Septemeber 2005. Cited on page 126.

OGF. The Open Grid Forum. URL http://www.gridforum.org. Last visited on January 27^{th}, 2012. Cited on page 19.

T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: Lessons in Creating a Workflow Environment for the Life Sciences. *Concurr. Comput. : Pract. Exper.*, 18:1067–1100, August 2006. Cited on pages 2, 20, 21, and 52.

OODT Website. Object-Oriented Data Technology, 2012. URL http://oodt.apache.org/. Last visited on January 17^{th}, 2012. Cited on page 17.

Oracle White Paper–Beginner's Guide to Oracle Grid Engine 6.2. Oracle Corporation, August 2010. Cited on page 16.

S. Pandey, W. Voorsluys, M. Rahman, R. Buyya, J. E. Dobson, and K. Chiu. A Grid Workflow Environment for Brain Imaging Analysis on Distributed Systems. *Concurrency and Computation: Practice and Experience*, 21(16):2118–2139, 2009. Cited on page 21.

P. Pantel, E. Crestan, A. Borkovsky, A.-M. Popescu, and V. Vyas. Web-Scale Distributional Similarity and Entity Set Expansion. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing (EMNLP'09)*, pages 938–947. Association for Computational Linguistics, August 2009. Cited on page 84.

M. P. Papazoglou and W.-J. Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16:389–415, July 2007. ISSN 1066-8888. Cited on page 50.

D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies . Technical report, Berkeley, CA, USA, 2002. URL http://www.ecst.csuchico.edu/~juliano/csci380/Papers/dPatterson2002roc.pdf. Cited on pages 38 and 39.

I. D. Peake and H. W. Schmidt. Systematic Simplicity-Accuracy Tradeoffs in Parameterised Contract Models. In *Seventh International ACM Sigsoft Conference on the Quality of Software Architectures (QoSA)*, Boulder, Colorado, USA, June 2011. Cited on page 30.

I. D. Peake, I. E. Thomas, and H. W. Schmidt. Typed formal concept analysis. In *Contributions to ICFCA 2009, Supplementary Proceedings, 7th International Conference on Formal Concept Analysis (ICFCA09)*, Darmstadt, Germany, May 2009. Verlag Allgemeine Wissenschaft. ISBN 3-935924-08-9. Cited on page 121.

W. W. Peterson and D. T. Brown. Cyclic Codes for Error Detection. *Proceedings of the Institute of Radio Engineers*, 49(1):228—235, 1961. Cited on page 92.

J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory Exclusion: Optimizing the Performance of Checkpointing Systems. *Software: Practice and Experience*, 29(2):125–142, 1999. ISSN 1097-024X. Cited on page 126.

R. H. Reussner, H. W. Schmidt, and I. H. Poernomo. Using Parameterised Contracts for Component Reliability Prediction. In H. Ehrig, B. Krämer, and A. Ertas, editors, *Integrated Design & and Process Technology*, volume IDPT 1. Society for Process & Design Sciences, 2002. Cited on page 63.

R. H. Reussner, H. W. Schmidt, and I. H. Poernomo. Reliability Prediction for Component-based Software Architectures. *Journal of Systems and Software*, 66:241–252, June 2003. ISSN 0164-1212. Cited on pages 30 and 40.

L. J. Rui Li, Z. Peng, Z. Yu, and C. Wang. Batch Text Similarity Search with MapReduce. *Lecture Notes in Computer Science*, 6612:412–423, 2011. Cited on page 84.

F. Salfner, M. Lenk, and M. Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42(3):10:1–10:42, 2010. ISSN 0360-0300. Cited on page 7.

H. W. Schmidt. Trustworthy Components: compositionality and prediction. *Journal of Systems and Software*, 65(3):215–225, March 2003. Cited on page 30.

H. W. Schmidt. Architecture-Based Reasoning About Performability in Component-Based Systems. In *Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '07, pages 130–137, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-69506-6. Cited on page 30.

M. Shaw and D. Garlan. *Software Architecture, Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN 0-13-182957-2. Cited on page 40.

D. P. D. Silva, W. Cirne, F. V. Brasileiro, and C. Grande. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In *Proceedings of International Euro-Par Conference*, pages 169–180, 2003. Cited on pages 3, 48, 49, 50, and 53.

D. B. Skillicorn, J. M. D. Hill, and W. F. Mccoll. Questions And Answers About BSP. *Scientific Programming*, 6(3):249—274, 1997. Cited on pages 44, 45, and 46.

W. J. Stewart. *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling*. Princeton University Press, Princeton, NJ, USA, 2009. Cited on pages 31, 32, 33, 34, and 35.

S. Suri and S. Vassilvitskii. Counting Triangles and the Curse of the Last Reducer. In *Proceedings of the 20th International Conference on World Wide Web (WWW'11)*, pages 607–614. ACM, March 2011. Cited on page 84.

C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. ISBN 0201745720. Cited on page 30.

O. Tatebe, K. Hiraga, and N. Soda. Gfarm grid file system. *New Generation Comput.*, 28(3): 257–275, 2010. Cited on page 16.

I. Taylor, M. Shields, I. Wang, and A. Harrison. The Triana Workflow Environment: Architecture and Applications. In I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 320–339. Springer London, 2007. Cited on page 21.

Y. M. Teo and X. B. Wang. ALiCE: A Scalable Runtime Infrastructure for High Performance Grid Computing. In *Proceedings of IFIP International Conference on Network and Parallel Computing, in Springer-Verlag, Lecture Notes in Computer Science Series 3222*, pages 101–109, 2004. Cited on page 16.

D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurr. Comput. : Pract. Exper.*, 17:323–356, February 2005. ISSN 1532-0626. Cited on pages 13 and 16.

The Open MPI Development Team. Open MPI: Open Source High Performance Computing, July 2011. URL http://www.open-mpi.org. Cited on page 85.

L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33 (8), August 1990. Cited on page 44.

L. G. Valiant. A Bridging Model for Multi-core Computing. *Journal of Computer and System Sciences*, 77(1), January 2011. Cited on page 44.

J. F. Wakerly. *Digital Design: Principles and Practices*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 3rd edition, 2000. ISBN 0130555207. Cited on page 91.

S. Wang and E. Rundensteiner. Scalable Stream Join Processing with Expensive Predicates: Workload Distribution and Adaptation by Time-Slicing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 299–310, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-422-5. Cited on pages 4 and 92.

W.-L. Wang, D. Pan, and M.-H. Chen. Architecture-Based Software Reliability Modeling. *Journal of Systems and Software*, 79:132–146, January 2006. ISSN 0164-1212. Cited on page 30.

J. Welkowitz, B. H. Cohen, and R. B. Lea. *Introductory Statistics for the Behavioral Sciences*. John Wiley & Sons, Inc., 7th edition, 2012. Cited on page 119.

S. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA 95)*, page 2436. ACM, 1995. Cited on page 42.

J. Xu, P. Townend, N. Looker, and P. Groth. FT-Grid: A System for Achieving Fault Tolerance in Grids. *Concurrency and Computation: Practice & Experience*, 20(3):297–309, 2008. Cited on pages 2, 48, and 50.

YourKit Website. YourKit Java Profiler, 2012. URL http://www.yourkit.com/. Last visited on March 14^{th}, 2012. Cited on page 126.

J. Yu and R. Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. *SIGMOD Rec.*, 34:44–49, September 2005. ISSN 0163-5808. Cited on page 22.

I. I. Yusuf. Recovery-Oriented Software Architecture for Grid Applications. In *Supplemental Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*, 2010. Cited on page 158.

I. I. Yusuf, H. W. Schmidt, and I. D. Peake. Evaluating Recovery Aware Components for Grid Reliability. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*, pages 277–280. ACM, 2009. ISBN 978-1-60558-001-2. Cited on page 158.

I. I. Yusuf, H. W. Schmidt, and I. D. Peake. "architecture-based fault tolerance support for grid applications". In *Proceedings of the Seventh International ACM SIGSOFT Conference on the Quality of Software Architectures*, QoSA'11, pages 177–181. ACM, 2011. ISBN 978-1-4503-0724-6. Cited on page 158.

X. Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, and R. D. Schlichting. Fault-tolerant Grid Services using Primary-Backup: Feasibility and Performance. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 105–114, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8694-9. Cited on page 50.

Y. Zhang, A. Mandal, C. Koelbel, and K. Cooper. Combined Fault Tolerance and Scheduling Techniques for Workflow Applications on Computational Grids. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 244–251, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3622-4. Cited on page 51.

D. Zurell, U. Berger, J. S. Cabral, F. Jeltsch, C. N. Meynard, T. Mnkemller, N. Nehrbass, J. Pagel, B. Reineking, B. Schrder, and V. Grimm. The virtual ecologist approach: simulating data and observers. *Oikos*, 119(4):622–635, 2010. ISSN 1600-0706. doi: 10.1111/j.1600-0706.2009.18284.x. URL http://dx.doi.org/10.1111/j.1600-0706.2009.18284.x. Cited on page 120.

Index