Autonomic Workflow Management System for Grid Computing

by

Mustafizur Rahman

Submitted in total fulfilment of
the requirements for the degree of

Doctor of Philosophy

Department of Computer Science and Software Engineering
The University of Melbourne, Australia

August 2010

Autonomic Workflow Management System for Grid Computing

Mustafizur Rahman

Supervisor: Professor Rajkumar Buyya

Abstract

As many of the large-scale scientific applications executed on Grids are expressed as complex scientific workflows, workflow management has emerged as one of the most important Grid services in past few years. Scientific workflows can be defined as the aggregation of Grid application services, which are executed on distributed Grid resources in a well defined order to satisfy the specific requirements of users. A workflow management system is generally employed to define, manage and execute these workflows in world-wide Grid environment. However, the increasing scale complexity, heterogeneity and dynamism of Grid environment that includes networks, resources and applications have made such workflow management systems brittle, unmanageable and insecure. Autonomic computing provides a holistic approach for the design and development of systems/applications that can adapt themselves to meet the requirements of performance, fault tolerance, reliability, security, etc., without manual intervention.

Therefore, we aim to develop algorithms (i.e. workflow scheduling strategy, resource coordination scheme) that aid the workflow management systems to incorporate the properties of autonomic computing and exhibit the ability to reconfigure itself to the changes in Grid environment, discover, diagnose and react to the disruptions of workflow execution as well as monitor and optimize its performance automatically. To this end, we leverage a distributed hash table based Peer-to-Peer (P2P) overlay to build a logical structure for the resource organization. The scalable and self-organizing nature of the P2P overlay provides a seamless framework for Grid resources to automatically self-configure themselves in the event of unavoidable situations, such as resource join, leave or failure. Further, we develop a reputation based dependable scheduling algorithm that improves the reliability of workflow execution through proactive resource provisioning by taking into account the prior performance and behaviour of Grid resources.

Thus, this thesis makes several contributions towards improving the state-of-the-art of autonomic workflow management systems for Grid computing environment and particularly, towards advancing the area of Grid workflow scheduling. The major contributions are: (i) proposed a taxonomy of autonomic application management for Grid computing and surveyed existing Grid systems; (ii) developed a dynamic critical path based workflow scheduling algorithm that can adapt to changing Grid environment; (iii) designed an architecture for autonomic workflow management system in Grids according to the requirements identified in proposed taxonomy; (iv) devised a decentralized and cooperative workflow scheduling algorithm, utilizing a self-configuring P2P overlay structure with regards to resource discovery, coordination and overall system decentralization; and (v) leveraging the proposed autonomic workflow management architecture, developed a reputation-based dependable workflow scheduling technique to enable self-healing behavior in Grid workflow management system.

This is to certify that

(i) the thesis comprises only my original work,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is less than 100,000 words in length, exclusive of table, maps, bibliographies, appendices and footnotes.

Signature_____

Date_____

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Professor Rajkumar Buyya for his systematic guidance as well as constant persuasion and encouragement throughout my PhD candidature. His profound knowledge and expertise in this field and generosity have given me the audacity to explore new research directions and successfully complete this thesis. In particular, I appreciate his enthusiasm and endless amount of energy that inspired me to put maximum effort into research. I am also grateful to him for providing me with the opportunity to work in an excellent research group at Cloud Computing and Distributed Systems (CLOUDS) Laboratory.

I wish to express my gratitude to Srikumar Venugopal for being my Ph.D committee member and guiding me at the early stage of my research studies, Rajiv Ranjan for sharing his expertise in decentralized Grid systems and helping me with regards to simulations and experimentation, and Md Rafiul Hassan for sharing his knowledge and participating in many fruitful discussions with me.

I would also like to thank Manish Parashar (Professor of Electrical and Computer Engineering, Rutgers University, USA), Boualem Benatallah (Professor, Computer Science and Engineering, University of New South Wales, Australia), and Henry Bal (Professor, Faculty of Sciences, Vrije University, The Netherlands) for their valuable feedbacks on some of the works presented in this thesis. I thank Shawkat Ali (Senior Lecturer, School of Computing Sciences, Central Queensland University, Australia) for his advices regarding general research guidelines.

I have been privileged to visit few research labs during my PhD candidature, which indeed inspired and helped me to keep myself motivated for conducting research. Thus, I wish to acknowledge Software Engineering and Technology Lab of Infosys Technologies Limited (Bangalore, India), Computer Systems Group (Department of Computer Science, Vrije University, The Netherlands), and Service Oriented Computing Research Group (Department of Computer Science and Engineering, University of New South Wales, Australia) for hosting me on several occasions. I also thank Melbourne School of Graduate Research (the University of Melbourne, Australia) for giving me the opportunity to participate in many UpSkills seminars and workshops that really facilitated to develop new skills, expand my networks, and get directions for career building.

I am thankful to all the past and present members of the CLOUDS Lab. In particular, I thank Jia Yu, Chee Shin Yeo, Anthony Sulistio, Hussein Gibbins, Krishna Nadiminti, Marcos Dias de Assuno, Marco Netto, James Andrew Broberg, Christian Vecchiola, Rodrigo Calheiros, Suraj Pandey, Saurabh Garg, William Voorsluys, Mohsen Amini, Anton Beloglazov, and Michael Mattess for their help and comments on my work. Regular feedbacks provided during weekly group meetings of CLOUDS Lab have also contributed significantly towards enhancement of the ideas presented in this thesis. Special thanks to Mukaddim Pathan for proof-reading parts of my thesis work and giving valuable suggestions on many occasions.

I always loved to be engaged in various outdoor and sporting activities during my candidature. In this regard, I would like to thank my friends from the University of Melbourne: Jubaer Arif, Lenin Mehedy, Andreas Schutt, Sarana Nutanong, Adeel Zafar, and Atif Mehmood. I have had great time with them outside the research activities and enjoyed a number of barbeques, picnics, outdoor and sporting events. In addition, I also appreciate the support from my team mates at Melbourne University Hockey Club (Metro 4 West), while playing in Victorian Hockey League.

I acknowledge Australian Government and the University of Melbourne for providing me with the scholarship to pursue my doctoral studies. I am grateful to Department of Computer Science and Software Engineering (CSSE), Melbourne School of Engineering, IEEE Technical Committee for Scalable Computing (TCSC), and CLOUDS Lab for giving travel support to attend international conferences. This work was partially supported by Australian Research Council (ARC) discovery project grant. I also thank the administrative and technical staff members in the CSSE department and School of Engineering, specially Pinoo Bharucha, Julien Reid, and Rosanna Parissi for being very helpful at all times.

Finally, I am deeply thankful to my family and friends, specially my father, mother, brother, sister and sister-in-law for their inspiration and support during my research studies. Above all, I am indebted to the All Mighty for His divine blessings that has made it possible to complete this thesis successfully.

Mustafizur Rahman
Melbourne, Australia
August 2010.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

This chapter introduces the context of the research presented in this thesis. It starts with a high-level overview of the key concepts related to the research problem addressed in the thesis. Then the fundamental motivations behind this research are stated and the proposed solution to address the research challenges is briefly presented. The chapter finally ends with a discussion on the principal research contributions and the organization of this thesis.

1.1 Background

This section provides a brief discussion on the key technologies including Grid computing, workflow management and autonomic computing that form the foundation of this thesis.

1.1.1 Grid Computing

Over the last two decades, Grid computing [43] has emerged as one of the most promising technologies to build high performance distributed computing infrastructures. Computational Grids enable the sharing, selection, and aggregation of geographically distributed heterogeneous resources, such as computational clusters, supercomputers, storage devices, and scientific instruments. These resources are under control of different Grid organizations, offer huge computational power, and being utilized to solve many impor-

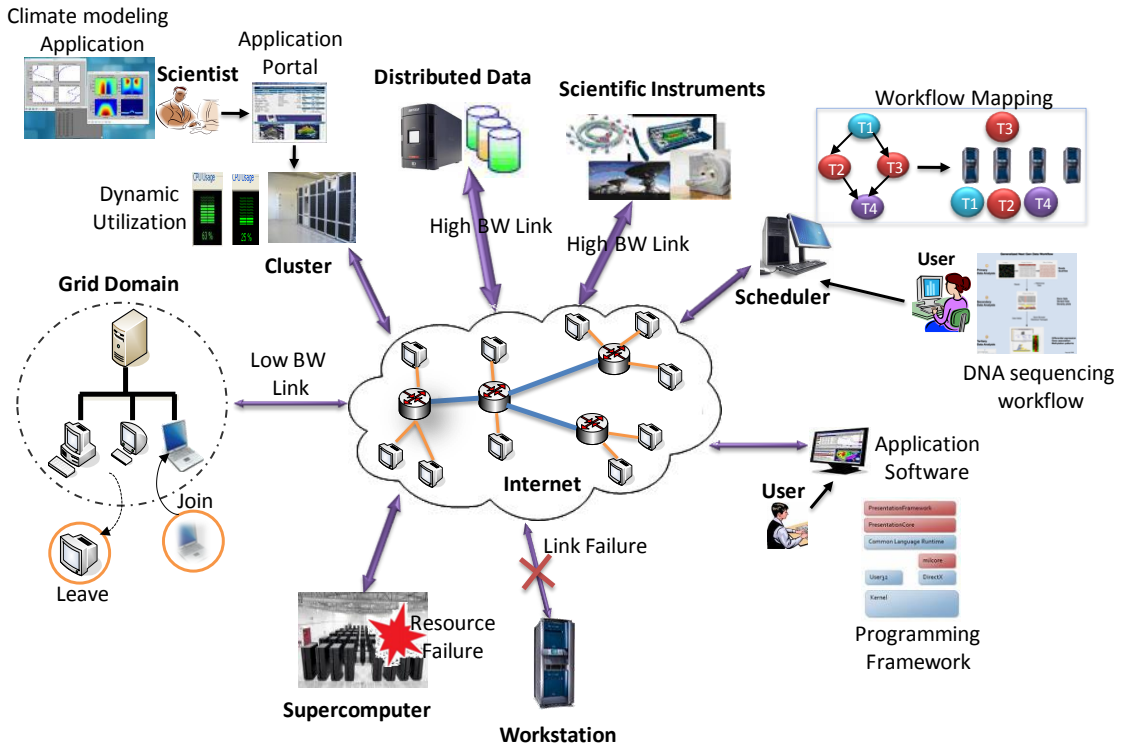


Figure 1.1: Heterogeneous and dynamic Grid computing environment.

tant scientific, engineering and business problems, such as protein folding, drug discovery, weather forecasting, earthquake engineering, financial modeling, and multi-player gaming. In general, Grid infrastructures are distributed, large, heterogeneous, uncertain, and highly dynamic.

Computational Grids can be categorized into different types, such as organization Grids, enterprise Grids, and global Grids depending on the scope of resource sharing. global Grids provide means to connect the Internet-wide distributed Grid resources, administered by multiple Grid sites with the aim of creating a collaborative computing environment. A sample scenario of resource sharing and application management in such a global Grid environment is shown in Fig. 1.1, where application scientists and users from different Grid sites/domains share various types of resources distributed over the world. The applications executed in such environment are generally computation or data intensive and composed of multiple tasks, where a task is a set of instructions that can be executed on a single processing element of a computing resource. Based on the dependency or relationship among these tasks, Grid applications can be divided into three

types: bag-of-task, tightly coupled message passing, and workflow applications. This thesis presents a framework to facilitate efficient scheduling and management of workflow applications in global Grids.

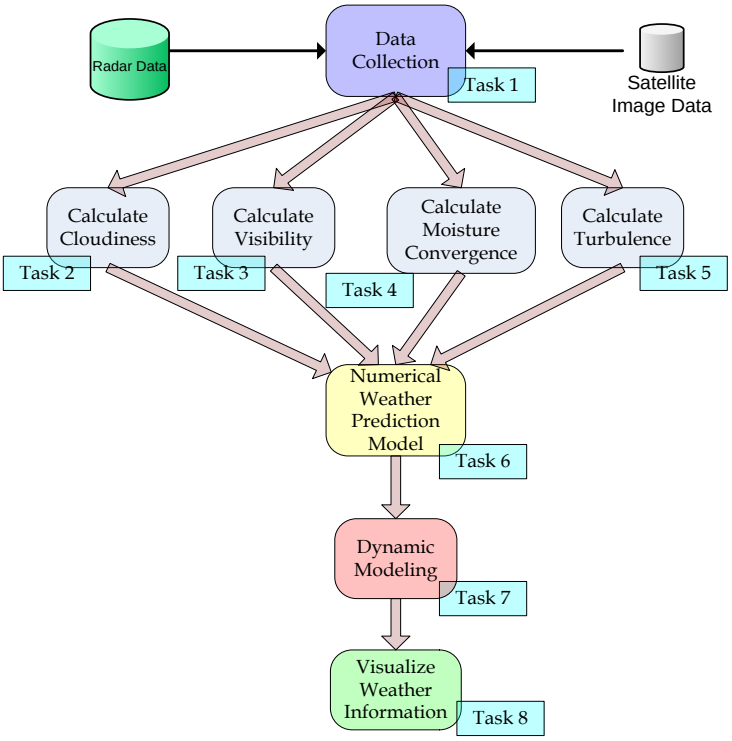
1.1.2 Workflow Management

Many of the large-scale scientific applications executed on the present-day Grids are expressed as complex e-Science workflow [73] [18], which is a set of ordered tasks that are linked by data dependencies (refer to Fig. 1.2(a)). A Workflow Management System (WMS) [133] is generally employed to define, manage, and execute these workflow applications on Grid resources. Due to the continuous demand of leveraging high performance computing infrastructure for the execution of workflow applications by the scientist and research community, WMS has emerged as one of the most important Grid services in past few years. Fig. 1.2(b) illustrates the execution of the workflow shown in Fig. 1.2(a) on a traditional distributed computing environment.

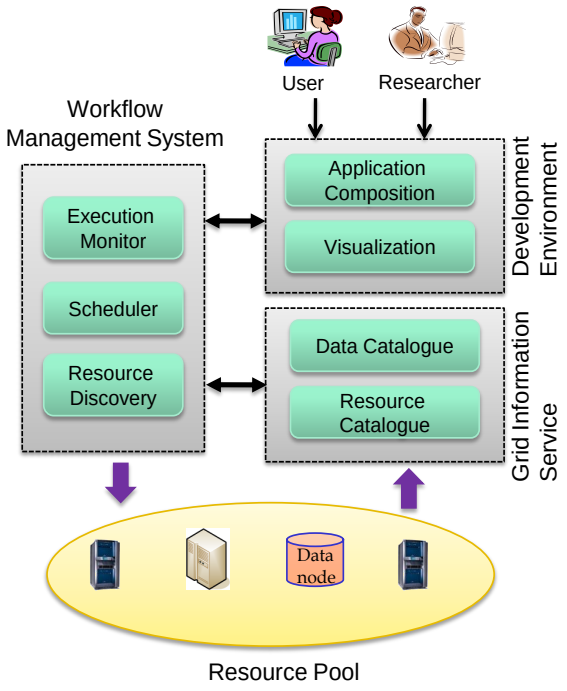
A WMS uses a specific scheduling strategy for mapping the tasks in a workflow to suitable Grid resources in order to satisfy user requirements. Realizing a WMS in global Grids requires a number of challenges to be overcome, which include workflow application modeling, resource discovery, task scheduling, information services, data management, and failure handling. However, the increasing scale complexity, heterogeneity, and dynamism of Grid environment have made such WMS brittle, unmanageable, and insecure. This thesis presents a framework for effectively managing workflow applications by devising a set of scheduling algorithms addressing these requirements.

1.1.3 Autonomic Computing

Autonomic Computing (AC) [89] is an emerging area of research for developing large-scale, self-managing, complex distributed system. The vision of AC is to apply the principles of self-regulation and complexity hiding for designing complex computer-based systems. Thus, AC provides a holistic approach for the development of systems that can adapt themselves to meet requirements of performance, fault tolerance, reliability, security, Quality of Service (QoS) etc. without manual intervention.



(a) Example of workflow: weather prediction application



(b) Workflow management system

Figure 1.2: Typical scientific workflow applications management scenario in distributed computing environment.

An autonomic workflow management system leverages the concept of AC and is able to efficiently define, manage, and execute workflow applications in heterogeneous and dynamic Grid environment by continuously adapting itself to the current state of the system. Therefore, this thesis aims to design and develop algorithms and policies for building an autonomic workflow management system.

1.2 Motivation for Autonomic Workflow Management System

Generally, the users and application scientists are not much aware of the low level Grid infrastructure, such as programming environment, core middleware services, and runtime systems; but still opt for high confidence level in composition and deployment of their scientific applications, such as workflow. Thus, they rely on the third party workflow brokering or scheduling services, which abstract the underlying complexity of the system and facilitates efficient execution of the applications in Grid resources.

However, one of the major drawbacks involved with these workflow schedulers is that they rely on the centralized or semi-centralized hierarchical resource information services, such as Monitoring and Discovering Service (MDS) [41]. Current studies [139] have shown that the existing centralized model for information services do not scale well as the number of users and resource providers increase in the system. Moreover, if the centralized links leading to these services fail then no scheduler in the system can undertake scheduling related activities due to the lack of up-to-date resource information.

Further, Grids are heterogeneous and dynamic environments consisting of computing, storage and network resources with different capability and availability. As a result, any scheduling decision that is based on the static resource information, would lead to generating inefficient schedule (mapping of tasks to resources) and degrade application performance. Thus, workflow schedulers should incorporate dynamic scheduling approaches that adapt to the changing resource conditions of Grids through just-in-time scheduling. In addition, the workflow schedulers adopting these dynamic scheduling algorithms are also required to cooperate with each other in order to avoid load balancing problem and

generate efficient schedules globally.

Next, uncertainty and unreliability are facts in a large scale heterogeneous Grid environment, which are triggered by multiple factors, including: (i) software and hardware failures as the system and application scale that lead to severe performance degradation and critical information loss; (ii) dynamism that results from temporal resource behaviours, which should be detected and resolved at runtime to cope with changing conditions; and (iii) lack of complete global knowledge that hampers efficient decision making as regards to composition and deployment of the application elements.

Autonomic computing has been emerged to cope with the aforementioned challenges by being context aware, adaptive, and resilient as well as providing agent based mechanisms for efficient decentralized cooperation and coordination. Thus, this thesis endeavors to explore autonomic management of workflow applications in global Grids, which implies the utilization of autonomic computing principles for the composition, deployment, and management of workflows in Grid computing environment.

1.3 Problem Description

This section presents the research problem and outlines the research questions to be addressed in this thesis. It also briefly describes the proposed methodology to solve the problem of Grid workflow management and answer these questions.

1.3.1 Research Hypothesis

The research hypothesis investigated in this thesis is:

Autonomic computing principles and algorithms for workflow management can effectively overcome the limitations of inherent uncertainty and dynamism of Grid environment as well as significantly improve the performance of workflow execution in global Grids.

1.3.2 Research Issues

The key research challenges that are required to be addressed in various aspects of autonomic workflow management are as follows.

Coordination of Workflow Schedulers:

- What kind of coordination mechanisms are required to be adopted to ensure effective interaction among the workflow schedulers, while maintaining scalability and flexibility?

- How can the workflow schedulers reconfigure themselves when policies and components are added or removed from the system due to changes in the environment?

System monitoring:

- How can the autonomic element in a Grid site identifies itself, discovers, and verifies the identities of other entities of interest dynamically, and establishes relationships with these entities to interact in a secure manner?

Failure management:

- How can workflow management systems diagnose and react to any disruptions, such as resource or task failure occurred during the execution of workflows?

- How can the system automatically anticipates, detects, identifies and protects against malicious attacks or cascading failures to maintain overall system security and integrity?

Runtime optimization:

- How can changes to the workflows at run time be identified, accommodated and the execution to be optimized accordingly?

1.3.3 Proposed Solution

In order to facilitate autonomic workflow management, we propose to develop algorithms (i.e. workflow scheduling strategy, resource coordination scheme) that aid the workflow schedulers to incorporate the autonomic computing features and exhibit the ability to reconfigure itself to the changes in the Grid environment, discover/diagnose and react to the disruptions of workflow execution as well as monitor and optimize its performance

automatically. To this end, we leverage a Distributed Hash Table (DHT) based Peer-to-Peer (P2P) overlay to build a logical structure for the resource organization. The scalable and self-organizing nature of the P2P overlay provides a seamless framework for Grid resources to automatically self-configure themselves in the event of unavoidable situations, such as resource join, leave or failure.

The proposed workflow scheduling approaches utilize the Grid-Federation [101] model in regards to resource organization and Grid networking. Grid-Federation aggregates the distributed application scheduling and resource provisioning services as part of a cooperative resource sharing environment. At every site in the federation, there is a Grid Federation Agent (GFA) that is responsible for managing the execution of applications submitted by the users (see Fig. 1.3). We extend the GFA to develop an autonomic element [61], called Grid Autonomic Scheduler (GAS). Every contributing Grid site maintains its own GAS service. A GAS service is composed of a software components, Grid Autonomic Manager (GAM) that is responsible for continuous monitoring and performance optimization through adapting.

In the proposed approach, workflow schedulers (GASs) post their resource demands by submitting a Resource Claim object to the DHT-based decentralized coordination space, while resource providers update the resource information by submitting a Resource Ticket object. These objects are mapped to the coordination space using a spatial hashing technique [116]. Once a resource ticket matches with one or more resource claims, the coordination space sends notification messages to the resource claimers in such a way that the corresponding resource ticket issuer is not overloaded. Thus, this mechanism enables the workflow schedulers to distribute the workload efficiently and optimize their scheduling efficiency through cooperative decision making using a virtual global shared space.

Further, we develop a reputation-based workflow scheduling technique based on the aforementioned P2P overlay to counter the effect of inherent unreliability and temporal characteristics of Grid resources. The scheduling algorithm considers reliability of a Grid resource as a statistical property, which is globally computed in the coordination space using dynamic feedbacks or reputation scores assigned by individual Grid service consumers. As a result, it facilitates Grid schedulers to achieve self-healing capability by

taking into account the Grid resources' prior performance and behaviour for facilitating opportunistic and reliable placement of workflow tasks.

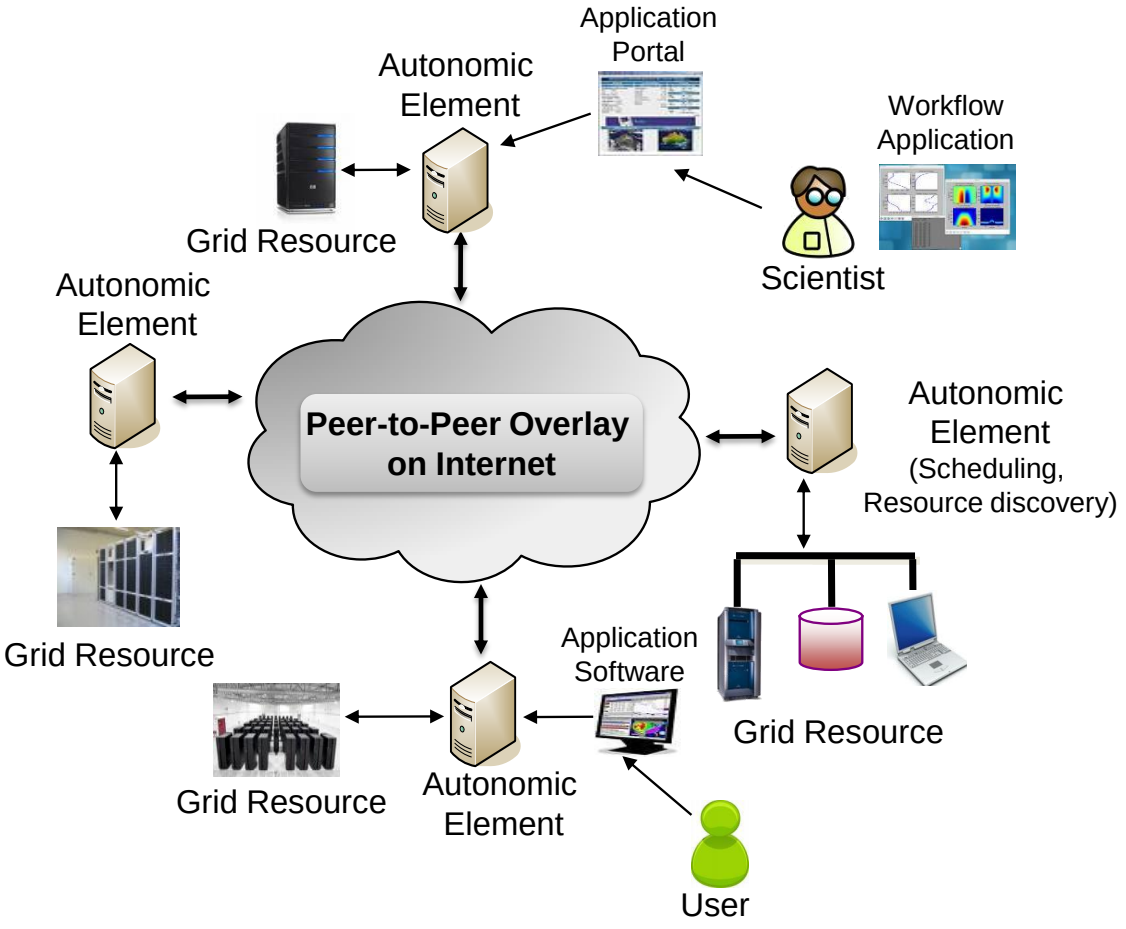


Figure 1.3: Proposed solution for autonomic workflow management.

For shared computing environments such as global Grids, simulation based techniques are suitable to do performance evaluation of self-managing algorithms and policies in a repeatable and controllable manner as users, resources, and application components are distributed across multiple organizations. Therefore, to appraise the proposed autonomic workflow management techniques with respect to different system and application scenarios, we perform experiments on a simulated global Grid environment, where the configuration of resources is modeled from existing real-world Grids including NorduGrid, AuverGrid, Grid5000 and LCG [62].

The simulation environment is modeled by combining two well known Java-based discrete event simulators: *GridSim* [22] and *PlanetSim* [49]. GridSim is a toolkit for

modeling and simulation of heterogeneous Grid systems. It provides a comprehensive facility for simulation of different types of resources, users, applications, scheduling policies, and fault management scenarios. On the other hand, PlanetSim is an event-based overlay network simulator. It can simulate both unstructured and structured P2P overlay networks. We also implement a prototype workflow management system utilizing the functionalities offered by Aneka Federation [99].

1.4 Contributions

This thesis makes several contributions towards improving the understanding of autonomic workflow management systems for Grid computing environment, particularly towards advancing the area of Grid workflow scheduling. The major contributions are as follows:

(i) This thesis provides a comprehensive taxonomy of autonomic application management in Grids that covers various aspects of application composition, application scheduling, coordination, monitoring as well as system characteristics and self-* properties. The proposed taxonomy is also mapped to various present-day Grid systems (specially, workflow management systems) not only to provide a basis for differentiating these systems but also to identify strengths and weaknesses of the state-of-art in Grid application (i.e. workflow) management. Thus, this thesis presents a roadmap for the researchers to interpret key concepts and to perform a comparative analysis of the related technologies in this domain.

(ii) This thesis discusses the workflow scheduling problem and describe the existing heuristic and meta-heuristic based workflow scheduling strategies in Grids. In addition, it also presents a dynamic critical path based workflow scheduling algorithm, which determines efficient mapping of workflow tasks to Grid resources by calculating the critical path in the workflow task graph at every step. The performance of the proposed algorithm, called DCP-G (Dynamic Critical Path for Grids) is evaluated against the existing approaches for different types and sizes of workflows through discrete-event simulations. The results demonstrate that DCP-G can naturally adapt to temporal resource behavior and avoid performance degradation in dynamically changing Grid environments.

(iii) This thesis introduces an architecture for autonomic workflow management system for Grids according to the requirements identified in proposed taxonomy. It also describes the system models, such as Grid model and coordination model to realize this architecture. To address the research problem in a practical context, this thesis provides the design and implementation of a prototype workflow management system utilizing the functions offered by Aneka Federation. The capabilities of the prototype system have been demonstrated using sample scientific workflow application.

(iv) This thesis proposes a decentralized and cooperative workflow scheduling technique utilizing a self-configuring P2P overlay structure with regards to resource discovery, coordination, and overall system decentralization. It also presents a decentralized resource provisioning algorithm for mapping the tasks to resources leveraging a DHT-based coordination space. Through extensive simulations, a sensitivity analysis of our scheduling technique is performed for the critical performance metrics, such as scheduling message complexity and makespan of the workflow. Further, this thesis also measures the effectiveness of the proposed approach along two dimensions (load balancing and coordination efficiency) as compared to non-cooperative scheduling and centralized coordination approaches.

(v) This thesis investigates the methods to enable self-healing behavior in Grid workflow management system. For this purpose, it presents a dependable workflow scheduling technique that aids the workflow scheduling entities in Grids to gain significant performance gains as compared to traditional approaches in the event of unsuccessful job execution or resource failure through reputation-based intelligent resource provisioning. The effectiveness of this contribution is appraised through a comprehensive simulation-driven analysis of the proposed approach based on realistic and well-known application failure model in order to capture the transient behaviours that prevail in existing Grid-based e-Science workflow execution environments. This thesis also performs a comparative evaluation that demonstrates the self-adaptability of the proposed approach in comparison to Grid environments, where: 1) resource behaviours do not change (i.e. no failure occurs), therefore no self-management is required; and 2) transient conditions exist but runtime systems and application elements have no capability to self-adapt.

1.5 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 presents an overview of autonomic computing system along with its properties. Then it provides a taxonomy of autonomic application management in Grids and a detailed survey of several representative Grid systems (specially, workflow management systems) to demonstrate the comprehensiveness of the taxonomy. The positioning of this thesis with respect to this taxonomy is also summarized at the end of this chapter.

Chapter 3 discusses the workflow scheduling problem and describes the existing heuristic and meta-heuristic based workflow scheduling strategies in Grids. In addition, this chapter also proposes a dynamic critical path based workflow scheduling heuristic that takes into account the dynamic behavior of Grid resources.

The thesis then concentrates on developing the design and architecture of an autonomic workflow management system in Chapter 4 according to the requirements identified in the taxonomy, presented in Chapter 2. The prototype implementation of the proposed system is also discussed in this chapter.

Chapter 5 and 6 are derived based on the dynamic workflow scheduling heuristics discussed in Chapter 3 and the autonomic workflow management architecture presented in Chapter 4. In particular, Chapter 5 proposes a decentralized and cooperative workflow scheduling approach that leverages a DHT-based self-configuring overlay for resource coordination and a cooperative decision making strategy to achieve runtime optimization. Furthermore, Chapter 6 introduces a reputation based dependable workflow scheduling technique to enable self-healing property for the workflow management system.

Finally, Chapter 7 concludes the thesis with a discussion of the main findings and a comprehensive future research roadmap. The relationship among the chapters is shown in Fig. 1.4.

1.6 Publication Record

Portions of the work presented in this thesis have been partially or completely derived from the following set of research papers published during the course of the PhD candi-

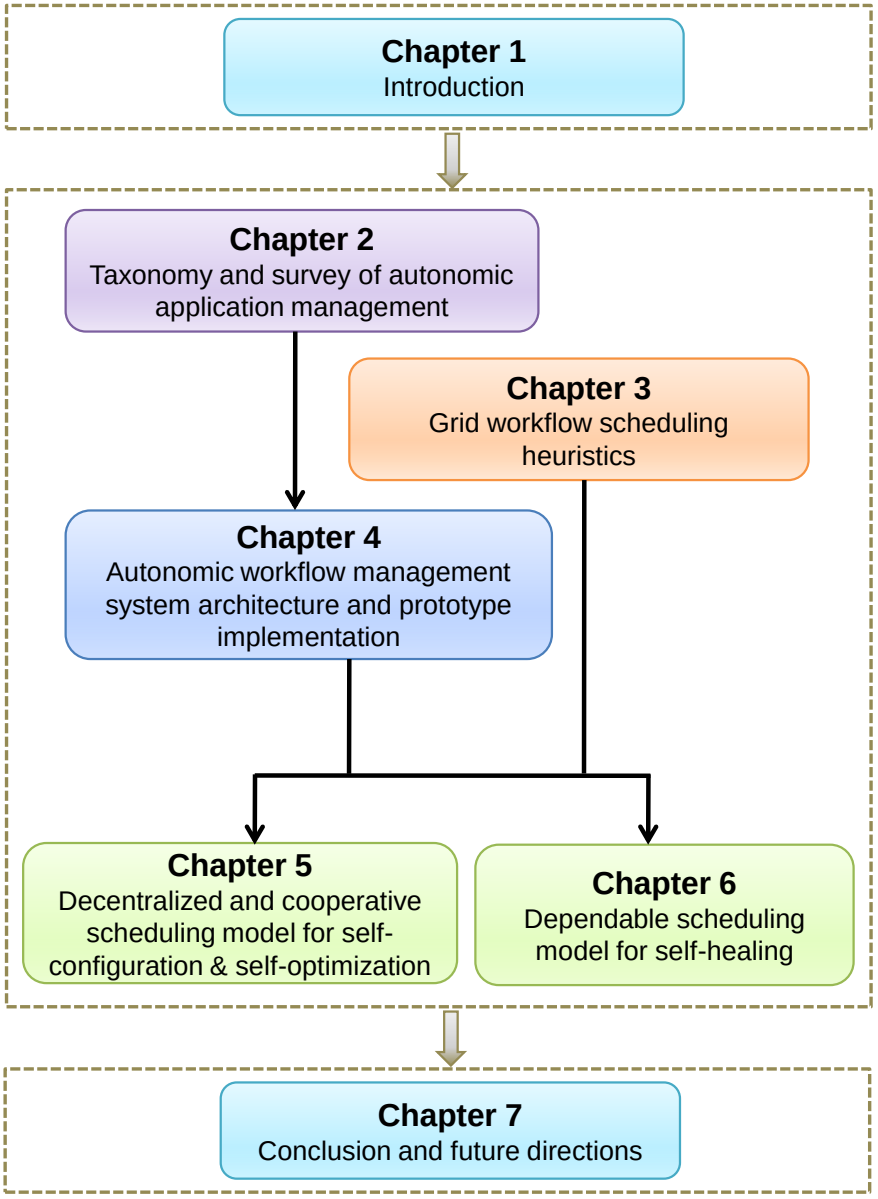


Figure 1.4: Organization of thesis chapters.

dature.

Chapter 1 is partially derived from the following publication.

- **Mustafizur Rahman** and Rajkumar Buyya, An Autonomic Workflow Management System for Global Grids, *In Proceedings of 2nd IEEE TCSC Doctoral Symposium, in conjunction with 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008)*, Lyon, France, May 2008.

Chapter 2 is partially derived from the following publications.

- **Mustafizur Rahman**, Rajiv Ranjan, and Rajkumar Buyya, Decentralized Management in Grid and Cloud Computing Systems: Challenges, Technologies and Opportunities, *Advancements in Distributed Computing and Internet Technologies: Trends and Issues*, S. Pathan et al. (eds.), IGI Global, USA, 2011.

- **Mustafizur Rahman**, Rajiv Ranjan, and Rajkumar Buyya, A Taxonomy of Autonomic Application Management in Grids, *In Proceedings of 16th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2010)*, Shanghai, China, December 2010.

- **Mustafizur Rahman**, Rajiv Ranjan, Rajkumar Buyya, and Boualem Benatallah, A Taxonomy and Survey on Autonomic Management of Applications in Grid Computing Environments, *Concurrency and Computation: Practice and Experience (CCPE)*, DOI: 10.1002/cpe.1734, Wiley Press, New York, USA, 2011.

Chapter 3 is partially derived from the following publications.

- **Mustafizur Rahman**, Srikumar Venugopaol, and Rajkumar Buyya, Dynamic Critical Path based Algorithm for Scheduling Scientific Workflow Applications on Global Grids, *In Proceedings of 3rd IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*, Bangalore, India, December 2007.

Chapter 5 is partially derived from the following publications.

- Rajiv Ranjan, **Mustafizur Rahman**, and Rajkumar Buyya, A Decentralized and Cooperative Workflow Scheduling Algorithm, *In Proceedings of 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008)*, Lyon, France, May 2008.

- **Mustafizur Rahman**, Rajiv Ranjan, and Rajkumar Buyya, Cooperative and Decentralized Workflow Scheduling in Global Grids, *Future Generation Computer Systems (FGCS)*, Volume 26, Number 5, Pages: 753-768, Elsevier Press, Amsterdam, The Netherlands, May 2010.

Chapter 6 is partially derived from the following publications.

- **Mustafizur Rahman**, Rajiv Ranjan, and Rajkumar Buyya, Dependable Workflow Scheduling in Global Grids, *In Proceedings of 10th IEEE/ACM International Conference on Grid Computing (Grid 2009)*, Alberta, Canada, October 2009.

- **Mustafizur Rahman**, Rajiv Ranjan, and Rajkumar Buyya, Reputation based Dependable Scheduling of Workflow Applications in Peer-to-Peer Grids, *Computer Networks (COMNET)*, Volume 54, Number 18, Pages: 3341-3359, Elsevier Press, Amsterdam, The Netherlands, December 2010.

Chapter 2

An Overview of Autonomic Management of Grid Applications

In this chapter, we propose a taxonomy that characterizes and classifies different components of autonomic application management in Grids. We also survey several representative Grid systems developed by various projects world-wide to demonstrate the comprehensiveness of the taxonomy. The taxonomy not only highlights the similarities and differences of state-of-the-art technologies utilized in autonomic application management from the perspective of Grid computing, but also identifies the areas that require further research initiatives.

2.1 Introduction

Application management has emerged as one of the most important Grid services in past few years. An Application Management System (AMS) is generally employed to define, manage, and execute these scientific applications in Grid resources. However, the increasing scale complexity, heterogeneity, and dynamism of Grid environment that includes networks, resources, and applications have made such application management systems brittle, unmanageable, and insecure. Thus, leveraging the principles of autonomic computing [89] can help to efficiently manage applications in Grids by continuously adapting the system to the current state of the environment.

This chapter aims to survey the existing Grid systems that support autonomic application management. We classify these systems with respect to different aspects of autonomic application management, such as application composition, scheduling, monitoring, coordinating, and failure handling as well as how the self-management properties (self-configuring, self-optimizing, self-healing, and self-protecting) have been implemented or incorporated in these systems.

2.2 History of Autonomic Computing

Autonomic Computing (AC) is a self-managing computing model, and the word autonomic is derived from its biological origins. The control in human body works in a fashion (self-regulating) that usually no humans' interference and consciousness are required. Likewise, the goal of AC is to create systems that run themselves and are capable of high-level functioning, while keeping the system's complexity invisible to the user. In this section, we briefly illustrate the introduction of AC in the field of Information Technology and describe its evolution along the path of twenty first century's technological revolution.

2.2.1 Integrating Biology and Information Technology

The term Autonomic Computing is named after and patterned on the human body's Autonomic Nervous System (ANS) [89]. ANS is the cornerstone to our ability to perceive, adapt to and interact with the world around us; thus helping human beings to manage dynamically changing and unpredictable circumstances. It acts as a control system functioning largely below the level of consciousness and handles human body's management of breathing, digestion, salivation, fending off germs and viruses etc. as shown in Fig. 2.1.

Inspired by the functionalities of ANS, AC has emerged to equip computing systems with the self-managing mechanisms of human body achieved through ANS. An Autonomic Computing System (ACS) would manage and control the functioning of computing systems and applications without any user input or intervention, in the same way ANS regulates human body systems without conscious input from the individual. Similar to ANS, ACS constantly checks and monitors its external and internal environment as

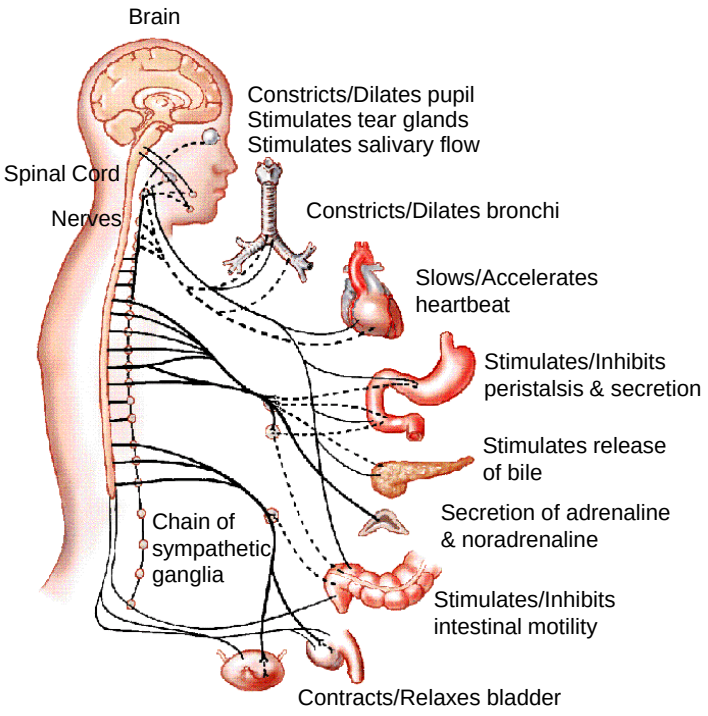


Figure 2.1: Autonomic Nervous System of human body.

well as automatically adapts to changing conditions in order to manage, optimize, repair, and protect itself.

2.2.2 Evolution of Autonomic Computing

In order to address the growing heterogeneity, complexity, and demand of computer systems, researchers of several organizations took initiatives to develop autonomous and self-managing systems in early 1990s, and it continued throughout the whole decade. These research initiatives gradually became matured and eventually facilitated autonomic computing to be emerged as an area of research.

Similar to the birth of Internet, one of the notable preliminary self-managing projects has been initiated by Defense Advanced Research Projects Agency (DARPA) in 1997 for a military application [6]. The project was called Situational Awareness System (SAS), and its aim was to create personal communication and location devices for soldiers in battlefield. As an outcome, soldiers had been able to enter status report (i.e. discovery of enemy tanks) into their personal device. This information would automatically spread

over to all other soldiers based on a decentralized peer-to-peer mobile adaptive routing. The latest status report could then be called up accordingly, while entering an enemy area.

Further, DARPA initiated another project related to self-management, named Dynamic Assembly for Systems Adaptability, Dependability and Assurance (DASADA) [3]. The objective of the DASADA program was to research and develop technology that would enable mission critical systems to meet high assurance, dependability, and adaptability requirements. Essentially, it dealt with the complexity of large distributed software systems and pioneered the architecture-driven approach to self-management [59].

NASA utilized the features of autonomous systems in late 1990s for its space projects, such as DS1 (Deep Space 1) and Mars Pathfinder [82]. In particular, NASA's interest was to make its deep-space probes more autonomous so that the probes could quickly adapt to extraordinary situations. To address this challenge, NASA designed a Remote Agent architecture, where the Remote Agent integrated constraint-based temporal planning and scheduling, robust multi-threaded execution, and model-based mode identification and reconfiguration. Thus, spacecrafts were able to carry out autonomous operations for long periods of time with no human intervention.

IBM started the autonomic computing initiative in 2001 with the ultimate aim to develop computing systems capable of self-management so that it could overcome the rapidly growing complexity of computing systems management [48]. On March 8, 2001, IBM Senior Vice President and Director of Research Dr. Paul Horn presented the importance and direction of autonomic computing during a keynote speech in the National Academy of Engineering conference at Harvard University [58]. He suggested that complex computing systems should be able to independently take care of the regular maintenance and optimization tasks; thus reducing the workload on the system administrators. Shortly after, IBM Server Group introduced the Server's Group project with codename eLiza. Eventually, *Project eLiza* became known as the autonomic computing project.

In 2003, IBM introduced architectural blueprint to build Autonomic Computing System [61]. In that blueprint, IBM proposed the architectural concept of autonomic computing and described five building blocks for an autonomic system. It also outlined the four properties of an autonomic (i.e. self-managing) system namely, self-configuring, self-optimizing, self-healing, and self-protecting. These properties are described in detail in

Section 2.3.1.

Gradually, IBM became the leader in the autonomic computing space and offered many effective self-managing software toolkits, such as Tivoli. In 2005, IBM launched a set of new development tools for Autonomic Computing with the hope that these will help pave the way for increased mainstream adoption of autonomic computing. The release included an autonomic management tool called, Policy Management for Autonomic Computing (PMAC) that makes decisions based on policies or business rules created by the developers when embedded within software applications.

In 2005, Parashar et al. [90] addressed the constant growth and increasing scale complexity of dynamic and heterogeneous components in computational Grids, and proposed to utilize the autonomic computing features for the composition, deployment, and management of complex applications in such environment. As part of this initiative, they introduced project Automate [91], which provides a framework for enabling autonomic application management in Grids.

Recently, the world has seen a paradigm shift in the consumption and delivery of IT services with the emergence of Cloud Computing. The computational Clouds address the explosive growth of Internet-wide computational/storage devices, and provides a superior user experience through scalability, reliability, and utility. The principles of autonomic computing have also been utilized in computational Clouds [68].

The emergence and evolution of autonomic computing from its origin, autonomous systems can be realized in brief by Table 2.1.

2.3 Overview of Autonomic Computing Systems

An autonomic computing system makes decisions on its own using high-level policies in order to achieve a set of goals. It constantly checks, monitors, and optimizes its status, and automatically adapt itself to the changing conditions. As widely reported in literature [61], an ACS is composed of Autonomic Elements (AE) interacting with each other. AE is the basic building block of ACS, and it can be considered as a software agent. An AE consists of one Autonomic Manager (AM) and one or more Managed Element (ME). The core component of AE is a control loop that integrates AM with ME (refer to Fig. 2.2).

Table 2.1: Timeline of Autonomic computing evolution

Year	Term	Description
1997	Situational Awareness System (SAS)	DARPA initiated SAS project to provide the soldiers real-time situational awareness information.
1998	Autonomous Agent	NASA made its deep-space probes more autonomous.
2000	DASADA	DARPA introduced gauges and probes in the architecture of software systems for monitoring the system.
2001	Autonomic Computing	IBM pushed Autonomic Computing.
2003	AC Blueprint	IBM introduced architectural blueprint to build autonomic computing system.
2005	AC Development Tools	IBM offered new development tools for autonomic computing.
2005	Autonomic Grid Computing	The concept of autonomic Grid computing was proposed.
2009	Autonomic Cloud Computing	Principles of autonomic computing was utilized in computational Clouds.

The functionality of this control loop is similar to the generic agent model proposed by Russell and Norvig [108], in which an intelligent agent perceives its environment through sensors and uses these percepts to determine actions to execute on the environment.

The Managed Element is a software or hardware component from the system, which is given autonomic behavior by coupling it with an AM. Thus, ME can be a web server or database, a specific software component in an application (e.g., the query optimizer in a database), the operating system, a cluster of machines in a Grid environment, a stack of hard drives, a wired or wireless network, a CPU or printer etc.

The autonomic manager is a software component that can be configured by system administrators using high level goals. It uses the monitored data from sensors and internal knowledge (i.e., rules) of the system to plan and execute the low level actions that are necessary to achieve these goals. The goals are usually expressed by event-condition-action policies (e.g., when 95% of Web servers' response time exceeds 2 seconds, and there are available resources, then increasing number of active Web servers) or utility function policies (e.g., increasing and distributing available resources among different servers so that the utility is maximized).

AM uses a manageability interface (i.e., sensors and effectors) to monitor and control MEs as well as a five component analysis and planning engine, MAPE-K (comprised of

Monitor, Analysis, Plan, Execute, and Knowledge base) to manage self-managing activities. Sensors retrieve information regarding the current state of ME, and effectors execute the required actions set up by AM. Thus, sensors and effectors are linked together to create the control loop.

The Monitor observes the sensors, filters the data or system information collected by them (e.g. network/storage usage or CPU/memory utilization), and stores the relevant data in the Knowledge base. The Analysis engine compares the gathered data against the desired or expected values stored in Knowledge base. The Planning engine devises strategies to correct or adjust the trends identified by Analysis engine. Finally, the Execution engine carries out changes (e.g. adding/removing servers to a Web server cluster or changing configuration parameters in a Web server) to the ME through effectors and stores the affected values in the Knowledge base.

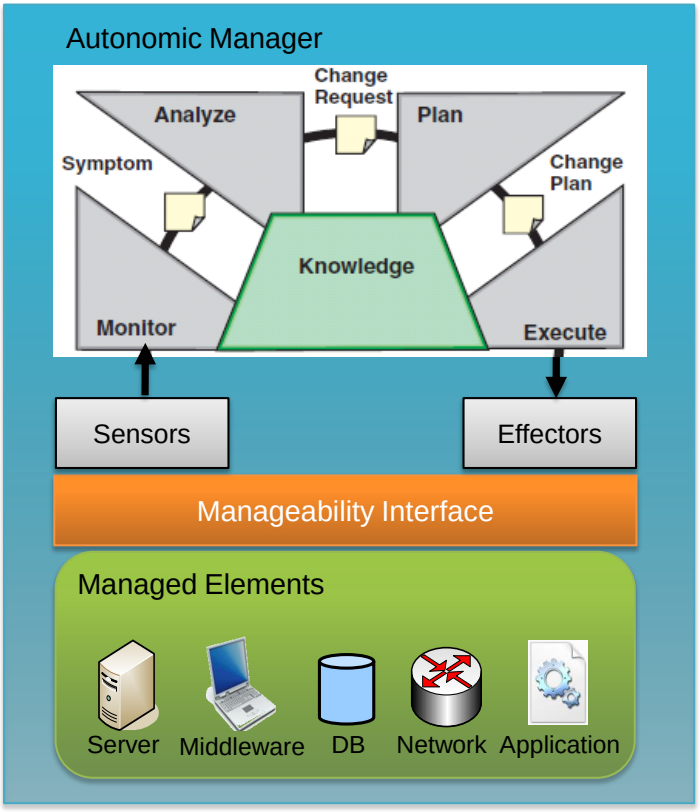


Figure 2.2: Architecture of an autonomic element.

2.3.1 Properties of Autonomic Computing Systems

Autonomic computing systems are generally composed of autonomic elements and capable of managing their behaviors and relationships with other systems in accordance with high-level policies. An autonomic system should possess at least eight key properties or characteristics. The primary four properties (refer to Fig. 2.3) of an autonomic system are called self-* properties, which are:

1. Self-configuring: An autonomic system must be able to automatically configure (i.e. setup) and reconfigure itself under dynamic and changing conditions.

2. Self-optimizing: An autonomic system must be able to optimize its working by monitoring the status quo and taking appropriate actions.

3. Self-healing: An autonomic system must be capable of identifying potential problems/failures and recovering from unexpected events that might lead the system to malfunction.

4. Self-protecting: An autonomic system must be capable of detecting and protecting itself from malicious attacks so as to maintain overall system security and integrity.

The secondary properties of autonomic systems are:

1. Self-awareness: An autonomic system requires to *know itself*, which can be achieved by having a detailed knowledge of its components and connections with other systems.

2. Context-awareness: An autonomic system should be aware of its execution environment by exposing itself and discovering other autonomic elements or systems in the environment.

3. Openness: An autonomic system should be able to function in a heterogeneous environment and be implemented on open standards and protocols.

4. Anticipatory: One critical property from the perspective of the users is that an autonomic system should be able to anticipate its needs and behaviours and act accordingly, while keeping its complexity hidden.

The description and example of the primary four properties of an autonomic system is presented in Table 2.2.

Figure 2.3: Self-* properties of autonomic computing system.

Table 2.2: Summary of self-* properties

Self-* Property	**Description**	**Example**
Self-configuring	Ability to adapt to changes in the system.	Installing software when it detects that some prerequisite software components are missing.
Self-optimizing	Ability to improve performance of the system.	Adjusting the current workload when it observes an increase or decrease in capacity.
Self-healing	Ability to discover, diagnose, and recover from faults.	Automatically re-indexing the files if a database index fails.
Self-protecting	Ability to anticipate, detect, identify, and protect against threats/intrusions.	Taking resources offline if it detects an intrusion attempt.

2.3.2 IBM Tivoli: A Case Study in Autonomic Computing

Tivoli [60] is a Systems Management Software toolkit provided by IBM that enables automation of routine management tasks for individual resource elements. It is designed based on CORBA-based architecture, which allows Tivoli to manage large number of

remote locations. Tivoli software tools are focused on various aspects of system management (e.g. security, storage, performance, availability, configuration, and operations) and facilitate provisioning of wide range of resources including systems, applications, middleware, networks, and storage devices.

The purpose of Tivoli platform is to bring self-managing c-apabilities into the IT infrastructure. The Tivoli software availability management portfolio provides tools to help customers monitor the health and performance of their IT infrastructure. Tivoli storage management tools help users to automatically and efficiently back up and protect data. The workload management tools use self-optimizing technology to optimize hardware and software use and verify if SLA goals are met successfully. Monitoring and event correlation tools help to determine when changes in the IT infrastructure require reconfiguration actions. These tools allow users to reconfigure their IT environment within minutes or hours rather than in days or weeks. The self-managing capabilities of Tivoli software toolkit are discussed in the following.

Self-configuring capabilities

In order to implement a self-configuring environment, Tivoli uses three software components: Configuration Manager, Storage Manager and Identity Manager. Tivoli Configuration Manager automatically configures to rapidly changing environments. It provides an inventory scanning engine and a state management engine that senses and detects when software on a target machine is out-of-synchronization with respect to a reference model for that class of machine. Tivoli Storage Manager provides self-configuring capabilities by automatically identifying and loading the appropriate drivers for the storage devices connected to the server. Tivoli Identity Manager uses automated role-based provisioning for dynamic account creation for users.

Self-optimizing capabilities

Tivoli Service Level Advisor performs self-optimizing activity by preventing Service Level Agreement (SLA) breaches with predictive capabilities. Based on the analysis of historical performance data from Tivoli Enterprise Data Warehouse, it predicts when critical SLA thresholds could be exceeded in the future. Tivoli Workload Scheduler monitors

and controls the flow of work through the IT infrastructure, and it uses sophisticated algorithms to maximize throughput and optimize resource usage. Tivoli Storage Manager supports Adaptive Differencing technology that facilitates the backup archive client to dynamically determine efficient approaches for creating backup copies of just changed bytes, blocks or files, and delivering improved backup performance. Tivoli Business Systems Manager enables optimization of IT problem repairs based on business impact of outages.

Self-healing capabilities

Tivoli utilizes several tools for implementing self-healing environment. Tivoli Enterprise Console collects and compares error reports, derives root cause, and initiates corrective actions. Tivoli Switch Analyzer correlates network device (Layer 2 switch) errors to the root cause without user intervention. Tivoli NetView enables self-healing by discovering TCP/IP networks, displaying network topologies, monitoring network health, and gathering performance data. Tivoli Storage Resource Manager automatically identifies potential problems through scanning and executes policy-based actions to resolve allocation of storage quotas or space and provide application availability.

Self-protecting capabilities

Tivoli Storage Manager self-protects by automating backup and archival of enterprise data across heterogeneous storage environments. On the other hand, Tivoli Access Manage self-protects by preventing unauthorized access and using a single security policy server to enforce security across multiple file types, applications, devices, operating systems, and protocols. Tivoli Risk Manager enables self-protecting by assessing potential security threats and automating responses, such as server reconfiguration, security patch deployment and account revocation.

2.4 Autonomic Management of Applications in Grids

Computational Grids enable the sharing, selection, and aggregation of geographically distributed heterogeneous resources, such as computational clusters, supercomputers, stor-

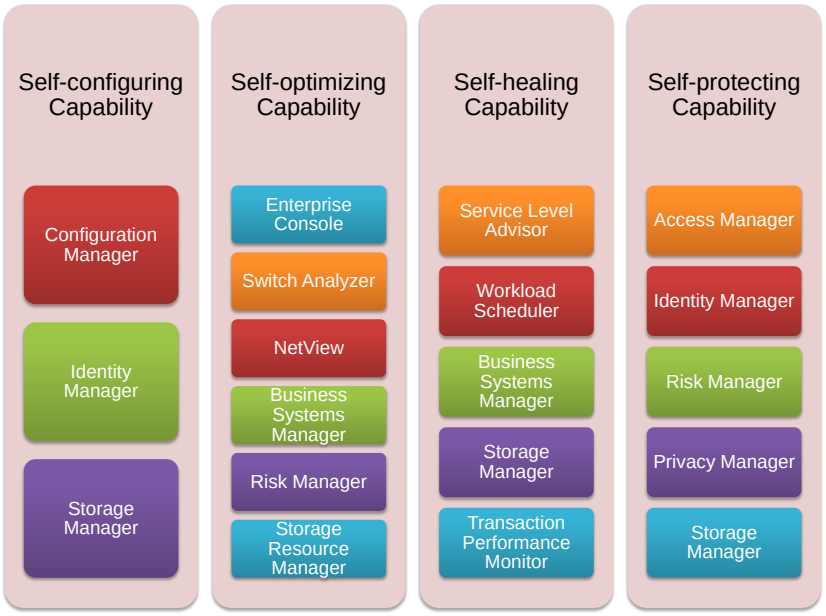


Figure 2.4: Components of IBM Tivoli enterprise management software toolkit.

age devices, and scientific instruments. These resources are under control of different Grid sites and being utilized to solve many important scientific, engineering, and business problems. In general, Grid infrastructures are distributed, large, heterogeneous, uncertain, and highly dynamic. A sample scenario of scientific application composition and management in such Grid environment is shown in Fig. 1.1, where application scientists and users from different Grid sites/domain share various types of resources distributed world wide.

Application scientists/users are not much concerned about the low level Grid infrastructure, such as runtime systems, programming environment and core middleware services, but still want high confidence level in composition and deployment of their scientific applications. They rely on the third party brokering or scheduling services, which abstract the underlying complexity of the system and facilitates efficient execution of the applications in Grid resources. However, many important applications in bioinformatics, medical imaging, and data mining require very accurate and reliable tools for conducting distributed experiments and analysis. The traditional Grid management methods, tools, and application composition techniques are inadequate to handle the dynamic interaction between the components and the sheer scale, complexity, uncertainty, and heterogeneity

of the Grid infrastructures as shown in Fig. 1.1.

Autonomic computing has been emerged to cope with the aforementioned challenges by being decentralized, context aware, adaptive, and resilient. Thus autonomic management of applications in Grids implies the utilization of autonomic computing features for the composition, deployment, and management of complex applications in Grids.

2.5 Taxonomy

In this section, we propose a taxonomy that categorizes and classifies the approaches of autonomic application management in the context of computational Grids with respect to the key features of AC. As shown in Fig. 2.5, it consists of six elements of autonomic application management: (1) application composition, (2) application scheduling, (3) coordination, (4) monitoring, (5) self-* property, and (6) system characteristics. In this section, we discuss each element and its classification in detail.

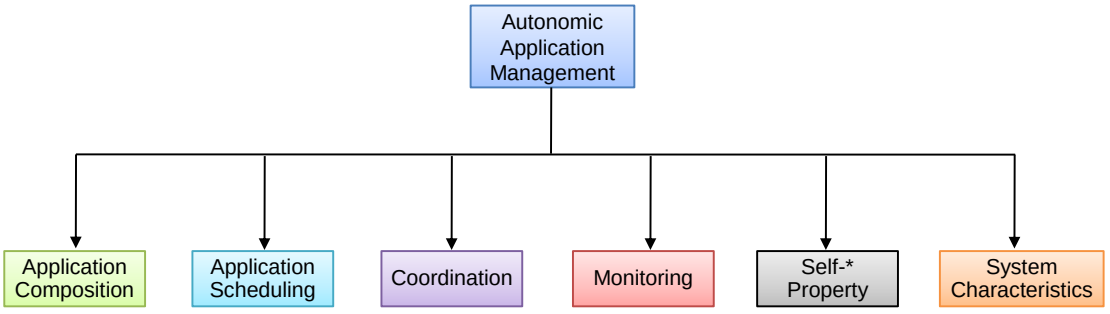


Figure 2.5: Elements of autonomic application management.

2.5.1 Application Composition

The applications executed in a distributed computing environment, such as Grids are generally computation or data intensive and users can experience better performance if they are able to execute these applications in parallel. In order to facilitate autonomic application management, the applications need to be composed dynamically based on the system configuration and users' requirements [30]. As shown in Fig. 2.6, application compo-

sition in a computational Grid is characterized by four factors: (a) application type, (b) application domain, (c) application definition, and (d) data requirement.

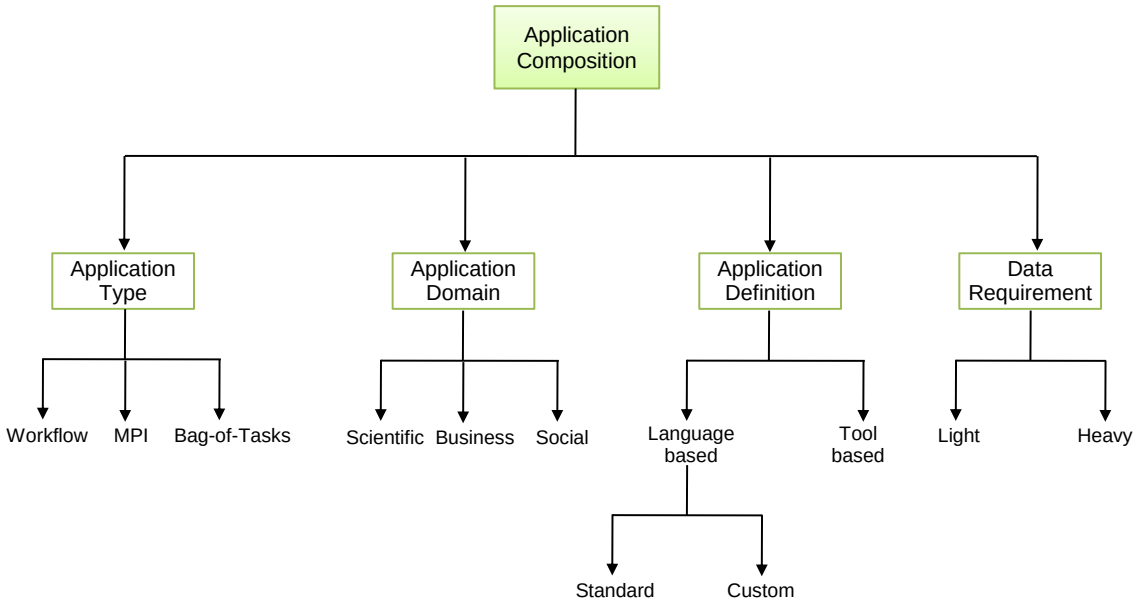


Figure 2.6: Application composition taxonomy.

Application Type

An application is composed of multiple tasks, where a task is a set of instructions that are executed on a single processing element of a computing resource. Based on the dependency or relationship among these tasks, Grid applications can be divided into three types: Bag-of-task (BOT), Message Passing Interface (MPI) and Workflow.

A BOT application [31] consists of multiple independent tasks with no communication among each other. The final result or output of executing the BOT application is achieved once all these tasks are completed. On the other hand, MPI applications [84] are composed of multiple tasks, where inter-task communication is developed with the Message Passing Interface (MPI) libraries. Since the tasks in most MPI applications need to communicate with each other during execution, the necessary processing elements are required to be available at the same time to minimize application completion time.

Finally, a workflow application can be modeled as a Directed Acyclic Graph (DAG), where the tasks in the workflow are represented as nodes in the graph and the depen-

dencies among the tasks are represented as the directed arcs among the nodes [97]. In a workflow, a task that does not have any parent task is called entry task and a task that does not have any child task is called exit task. A child task cannot be executed until all of its parent tasks are completed. The output of a workflow application is achieved when the exit tasks finish execution.

Application Domain

Grids offer a way to solve challenging problems by providing a massive computational resource sharing environment of large-scale, heterogeneous and distributed IT resources. With the advent of Grid technologies, scientists and engineers are building more and more complex applications to manage and process large scale experiments. These applications are spanned across three domains: scientific, business and social.

Many scientific (also known as e-Science and e-Research) applications, such as Bioinformatics, Drug discovery, Data mining, High-energy physics, Astronomy, and Neuroscience have been benefited with the emergence of Grids. Enabling Grids for E-sciencE (EGEE) [74] is considered as one of the biggest initiatives taken by European Union to utilize Grid technologies for scientific applications. Likewise, Business Experiments in GRID (BEinGRID) [1] is also the largest project for facilitating business applications, such as Business process modeling, Financial modeling and forecasting using Grid solutions. Recently, the emergence and upward growth of various social applications, such as Social networking have been widely recognized and the scalability of distributed computing environment is being leveraged for the better performance of these type of applications.

Application Definition

In general, users can define applications using definition languages or tools. In terms of definition language, markup language, such as Extensible Markup Language (XML) [4] is widely used specially for workflow specification as it facilitates information description in a nested structure. Therefore, many XML-based application definition languages have been adopted in Grids. Some of these languages, such as WSDL [7] and BPEL [64] have been standardized by the industry and research community (i.e. W3C [8]), whereas

some of them, such as xWFL [134] and AGWL [40] are customized according to the requirements of the system.

Although language-based definition of applications is convenient for expert users, it requires users to learn a lot of language-specific syntax. Thus, the general users prefer to use Graphical User Interface (GUI) based tools, such as Petri Nets [92] for application definition, where the application composition is better visualized. However, this graphical representation is later converted into other forms for further manipulation.

Data Requirements

Managing applications in Grids also needs to handle different types of data, such as input data, backend databases, intermediate data products, and output data. Many Bioinformatics applications often have small input and output data but rely on massive backend databases that are queried as part of task execution. On the other hand, some Astronomy applications generate huge output data that are feed into other applications for further processing. Some applications also need the data to be streamed between the tasks for efficient execution.

Thus, the data requirements of an application can be categorized into two types: light and heavy. If an application needs huge amount of data as input or generates massive intermediate or output data products then its data requirement is considered as heavy. These applications are generally known as data-intensive applications. Whereas, if the application is computation-intensive, it does not need much data to be handled and its data requirement is considered as light.

2.5.2 Application Scheduling

Effective scheduling is a key concern for the execution of performance driven Grid applications. Scheduling is a process of finding the efficient mapping of tasks in an application to the suitable resources so that the execution is completed with the satisfaction of objective functions, such as execution time minimization, as specified by Grid users. In this section, we discuss application scheduling taxonomy from the perspective of (a) scheduling architecture, (b) scheduling objective, (c) scheduling decision, and (d) sched-

uler integration, as shown in Fig. 2.7.

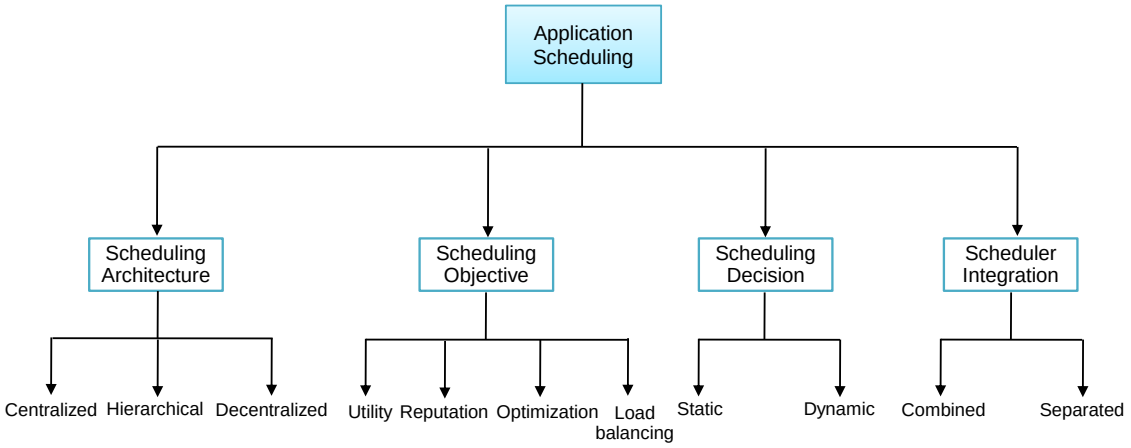


Figure 2.7: Application scheduling taxonomy.

Scheduling Architecture

The architecture of scheduling infrastructure is very important with regards to scalability, autonomy and performance of the system [57]. It can be divided into three categories: centralized, hierarchical and decentralized.

In centralized scheduling architecture [134], scheduling decisions are made by a central controller for all the tasks in an application. The scheduler maintains all information about the applications and keeps track of all available resources in the system. Centralized scheduling organization is simple to implement, easy to deploy and presents few management hassles. However, it is not scalable with respect to the number of tasks and Grid resources.

For hierarchical scheduling, there is a central manager and multiple lower-level schedulers. This central manager is responsible for handling the complete execution of an application and assigning the individual tasks of this application to the low-level schedulers. Whereas, each lower-level scheduler is responsible for mapping the individual tasks onto Grid resources. The main advantage of using hierarchical architecture is that different scheduling policies can be deployed at central manager and lower-level schedulers [57]. However, the failure of the central manager results in entire system failure.

In contrast, decentralized scheduler organization [104] negates the limitations of centralized or hierarchical organization with respect to fault-tolerance, scalability and autonomy (facilitating domain specific resource allocation policies). This approach scales well since it limits the number of tasks managed by one scheduler. However, this approach raises some challenges in the domain of distributed information management, system-wide coordination, security, and resource provider's policy heterogeneity.

Scheduling Objective

The application schedulers generate the mapping of tasks to resources based on some particular objectives. Usually, the schedulers employ an objective function that takes into account the necessary objectives and endeavour to maximize the output. The most commonly used scheduling objectives in a Grid environment are utility, reputation, optimization and load balancing.

Utility is a measure of relative satisfaction. In a utility driven approach, the users or service consumers prefer to execute their applications within certain budget and deadline, whereas the resource providers tend to maximize their profits. Reputation refers to the performance of computing resources in terms of successful task execution and trustworthiness. Optimization is related to the improvement of performance with regards to application completion time or resource utilization. Load balancing is also a measure of performance, where the workload on the resources is distributed in such a way so that any specific resource is not overloaded.

Scheduling Decision

An application scheduler uses a specific scheduling strategy for mapping the tasks in an application to suitable Grid resources in order to satisfy user requirements. However, the majority of these scheduling strategies are static in nature [121]. They produce a good schedule given the current state of Grid resources and do not take into account changes in resource availability.

On the other hand, dynamic scheduling [95] is done on-the-fly considering the current state of the system. It is adaptive in nature and able to generate efficient schedules,

which eventually minimizes the application completion time as well as improves the performance of the system.

Scheduler Integration

The scheduler component in a Grid system provides the service of generating execution schedules that map the tasks in an application onto distributed computing resources considering their availability and user's requirements. It also keeps track of the status of the tasks being executed on these Grid resources.

The application scheduler can be deployed in a Grid environment as a scheduling or brokering service to be consumed by the Grid users utilizing the principles of Service-oriented Architecture (SOA). In this case, the scheduler component is deployed separately in a server and the users submit their applications to this service [126], where the individual task scheduling and submission are managed by the scheduler. On the other hand, scheduler can also be combined or integrated into the system at user's side so that the users are not required to connect another service for scheduling purposes.

2.5.3 Coordination

The effectiveness of autonomic application management in a distributed computing environment also depends on the level of coordination among the autonomic elements, such as application scheduler or resource broker, local resource management system and resource information service. Lack of coordination among these components may result in communication overhead, which eventually degrades performance of the system. In general, the process of coordination with respect to application scheduling and resource management in Grids involves dynamic information exchange between various entities in the system. In this section, we discuss the coordination taxonomy from the view of (a) decision making, (b) component integration, and (c) negotiation policy as illustrated in Fig. 2.8.

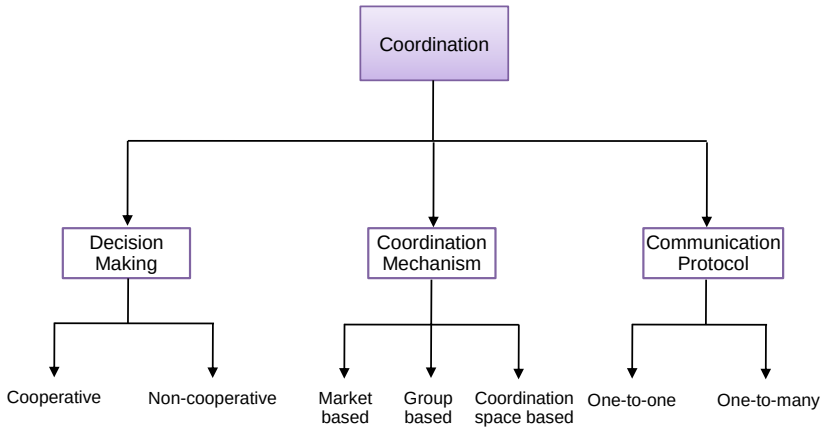


Figure 2.8: Coordination taxonomy.

Decision Making

In a distributed computing environment, the autonomic components communicate or interact with each other for the purpose of individual or system-wide decision making (e.g. overlay construction, task scheduling and load balancing). The process of decision making can be divided into two categories: cooperative and non-cooperative.

In the non-cooperative decision making scheme, application schedulers perform scheduling related activities independent of the other schedulers in the system. For example, Condor-G [47] resource brokering system performs non-cooperative scheduling by directly submitting jobs to the condor pools without taking into account their load and utilization status. This approach exacerbates the performance of the system due to load balancing and utilization problems.

In contrast, cooperative decision making approach [94] negotiates resource conditions first with the local site managers in the system, and if it fails, then negotiates with the other application level schedulers. Thus, it is able to not only avoid the potential resource contention problem but also distribute the workload evenly over the entire system.

Coordination Mechanism

Realizing effective coordination among the dynamic and distributed autonomous entities requires robust coordination mechanism and negotiation policies. Three types of coordination mechanisms are well adopted in Grids: market based coordination, group based

coordination and coordination space based coordination.

Market based mechanism views computational Grids as virtual marketplace in which economic entities interact with each other through buying and selling computing or storage resources. Typically, this coordination mechanism is used to facilitate efficient resource allocation. One of the common approaches to achieve market based coordination is to establish agreements between the participating entities through negotiations. Negotiation among all the participants can be done based on well-known agent coordination mechanism called contract net protocol [112]. In such mechanism, the resource provider works as a manager that exports its local resources to the outside contractors or resource brokers and is responsible for decision regarding admission control based on negotiated Service Level Agreements (SLA).

In collaborative Grid environment, resource sharing is often coordinated by the composition of groups (e.g. Virtual Organization (VO) [45]) of participating entities with similar interests. In Grids, VO refers to a dynamic set of individuals and institutions defined around a set of resource-sharing rules and conditions. The users and resource providers in a VO share some commonality among them, including common concerns, requirements and goals. However, the VOs may vary in size, scope, duration, sociology, and structure. Thus, inter-VO resource sharing in Grids is achieved through the establishment of SLA among the participating VOs.

Decentralized coordination space [75] provides a global virtual shared space for the autonomic elements in Grids. This space is concurrently and associatively accessed by all participants in the system, and the access is independent of the actual physical or topological proximity of the hosts. New generation DHT-based routing algorithms [115][107] form the basis for organizing the coordination space. The application schedulers post their resource demands by submitting a *Resource Claim* object into the coordination space, while resource providers update the resource information by submitting a *Resource Ticket* object. If there is a match between these objects, then the corresponding entities communicate with each other in order to satisfy their interests.

Communication Protocol

The interaction among the autonomic components is coordinated by the utilization of some particular communication protocols that can be divided into two types: One-to-one and One-to-many. Communication protocols based on One-to-many broadcast is simple but very expensive in terms of number of messages and network bandwidth usage. This overhead can be drastically reduced by adopting One-to-one negotiation among the resource providers and consumers through establishment of SLA.

2.5.4 Monitoring

Monitoring in Grids involves capturing information regarding the environment (e.g. number of active resources, queue size, processor load) that are significant to maintain the self-* properties of the system. This information or monitoring data is utilized by the MAPE-K autonomic loop and the necessary changes are accordingly executed by the autonomic manager through effectors. The sensing components of an autonomic element in Grids require appropriate monitoring data to recognize failure or suboptimal performance of any resource or service. As shown in Fig. 2.9, monitoring can be done in three levels: (a) execution monitoring, (b) status monitoring, and (c) directory service.

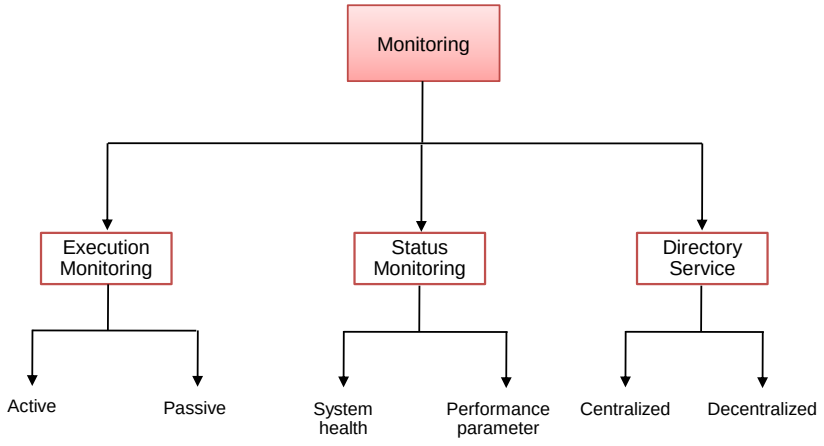


Figure 2.9: Monitoring taxonomy.

Execution Monitoring

Once an application is scheduled and the tasks in that application are submitted to corresponding Grid resources for execution, the scheduler needs to periodically monitor the execution status (e.g. queued, started, finished and failed) of these tasks so that it can efficiently manage unexpected events, such as failure. We identify two types of execution monitoring in Grids: active and passive.

The concept of active and passive monitoring of task execution in Grids is derived from the push-pull protocol [109], widely used in the research area of computer network. In active monitoring, information related to task execution is created by engineering the software at some levels, for example, modifying and adding code to implementation of the application or the operating system to capture functions or system calls so that the application itself can report monitoring data periodically (pulling). Moreover, the request for transferring status information is often initiated by the receiver or client in active monitoring strategy. For instance, the application scheduler can send an isAlive probe to the Grid resources currently executing its application for detecting the availability (e.g. online) of these resources at that time.

In contrast, passive monitoring technique captures status information at the resource or server side by the local monitoring service and reports monitoring data to the user or scheduler side periodically (pushing). For example, a resource provider in Grids can periodically inform the status of its system, such as current load to the interested application schedulers according to the requirements specified in the agreement between them.

Status Monitoring

The autonomic elements in Grids need to monitor the relevant system properties (i.e. system health) to optimize its operating condition and facilitate efficient decision making. System health data relates to runtime system information, such as memory consumption, CPU utilization, and network usage. The process of collecting system related information is straightforward and most of the operating systems provide a set of commands (e.g. top, vmstat in Linux) or tools to perform this operation.

In addition, the autonomic element also needs to monitor the performance parameters

of its operation, such as SLA violation if it incorporates market based mechanisms to interact with other autonomic elements in the system. To this end, it periodically measures the performance metrics (e.g. service uptime) with regards to the SLA and takes necessary steps to prevent the violation of agreement.

However, the monitoring service often suffers from the dilemma of deciding on how frequently and how much monitoring data should be collected to facilitate efficient decision making. Thus, dynamic and proactive monitoring approaches are essential in order to achieve autonomicity. For example, QMON [12] is an autonomic monitoring service that adapts its monitoring frequency and data volumes for minimizing the overhead of continuous monitoring, while maximizing the utility of the performance data.

Directory Service

The directory service provides information about the available resources in the Grid and their status, such as host name, memory size and processor load. The application schedulers or resource brokers rely on this information for efficient mapping of tasks in an application to the available resources. Based on the underlying structure, two types of directory services are available in Grids: centralized and decentralized.

In a centralized directory service (e.g. Grid Market Directory (GMD) [138]),monitoring data is stored in a centralized repository. Current studies [139] have shown that existing centralized models for resource directory services do not scale well as the number of users, brokers, and resource providers are increased in the system and are vulnerable to single point of failure. Whereas, decentralized information service distributes the process of resource discovery and indexing over the participating Grid sites so that load is balanced and if one site is failed, another site can take over its responsibility autonomously.

2.5.5 Self-* Property

In this section, we discuss the self-* properties taxonomy from the perspective of four primary properties: (a) self-configuring, (b) self-optimizing, (c) self-healing, and (d) self-protecting (refer to Fig. 2.10). As illustrated in Section 2.3.1, the evaluation of an autonomic system depends on to what extent it adopts or implements the self-* properties.

Further, its very difficult for a system to fully implement all the self-* properties and in many cases it becomes redundant. Thus most of the autonomic systems focus on some particular properties based on their requirements and goals.

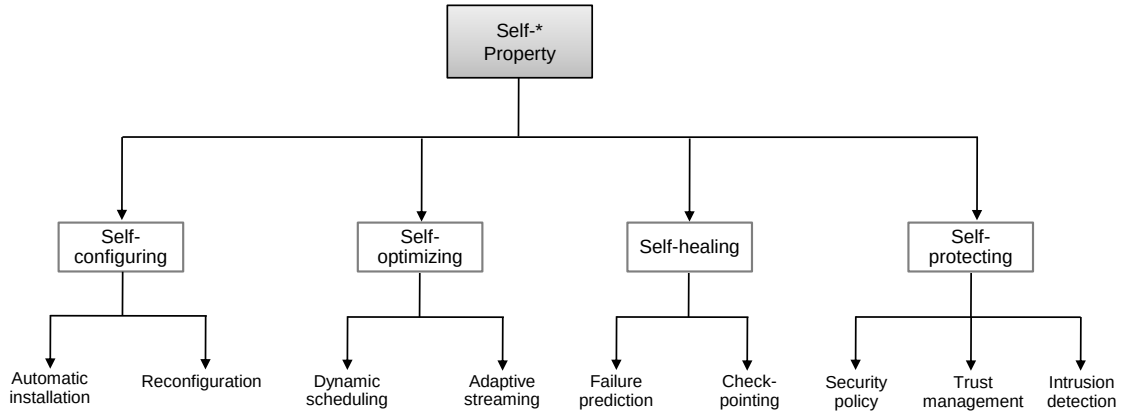


Figure 2.10: Self-* properties taxonomy.

Self-configuring

Self-configuration refers to the ability to adapt to changes in the system. In regards to autonomic application management, the system demonstrates self-configuration property by automatically installing the software components when it detects that some prerequisite components are outdated or missing. This ensures the timely update of the system without human intervention. In addition, the system also reconfigures itself in the event of dynamic and changing condition.

Self-optimizing

Self-optimization refers to the ability to improve performance of the system through continuous optimization. In Grids, the application management systems can use dynamic scheduling techniques for mapping application tasks to Grid resources in order to implement the self-optimizing property. The dynamic scheduling approach proactively monitors the status of the Grid resources and schedules tasks according to the current condition of the computational environment by dynamic policies, such as rescheduling. Another

continuous optimization strategy is utilization of adaptive streaming technique for data intensive applications [17].

Self-healing

Self-healing refers to the ability to discover, diagnose and recover from faults. Thus, the self-healing property enables a distributed computing system to be fault-tolerant by avoiding or minimizing the effects of execution failures. In a Grid environment, task execution failure can happen for various reasons: (i) sudden changes in the execution environment configuration, (ii) unavailability of required services or software components, (iii) overloaded resource conditions, (iv) system running out of memory, and (v) network failures. In order to efficiently handle these failures an autonomic system can adopt some preemptive policies, such as Failure prediction, Check-pointing, and Replication.

Failure prediction techniques [93] are used to predict the availability of the Grid resources continuously over certain period of time. Using these predictions, application schedulers plan the mapping of tasks to the resources considering their future availability in order to avoid possible task failures. The check-pointing technique [2] transfers the failed tasks transparently to other resources, so that the task continues its execution from the point of failure. The replication technique [10] executes the same task simultaneously on multiple Grid resources to increase the probability of successful task execution.

Self-protecting

Self-protecting refers to the ability to anticipate and protect against threats or intrusions. This property makes an autonomic system capable of detecting and protecting itself from malicious attacks so as to maintain overall system security and integrity.

Self-protecting application management can be achieved by implementing some proactive policies (i.e. dynamic access control) at both resource and user sides, such as providing accurate warning about potential malicious attack, taking networked resources offline if any anomaly is detected, and shutting down the system if any hazardous event occurs.

One technique for enabling self-protection is utilization of distributed trust management systems. A relevant distributed trust mechanism is PeerReview [38]. These distributed trust management systems determine malicious participants through behavioral

auditing. An auditor node A checks if it agrees with the past actions of an auditee node B. In case of disagreement, A broadcasts an accusation of B. Interested third party nodes verify evidence, and take punitive action against the auditor or the auditee.

Another approach to protect the system from malicious attacks is to leverage intrusion detection techniques [56], where the system performs online monitoring and analyzes the attacks/intrusions. Then a model is devised and trained using the past data that is used to successfully and efficiently detect future attacks.

2.5.6 System Characteristics

Due to its inherent nature, a distributed system possesses some characteristics, such as decentralization, heterogeneity, complexity, and reliability. These characteristics not only enforce challenges for designing the system but also make the system useful to the users. As indicated in Fig. 2.11, there are three characteristics of a distributed system that are related to autonomic application management: (a) complexity, (b)scalability, and (c) volatility. The more a system becomes complex or volatile, the more it needs to incorporate autonomic computing principles in order to avoid performance degradation and user dissatisfaction. Whereas, increasing the scalability of a system facilitates the adoption of autonomic features.

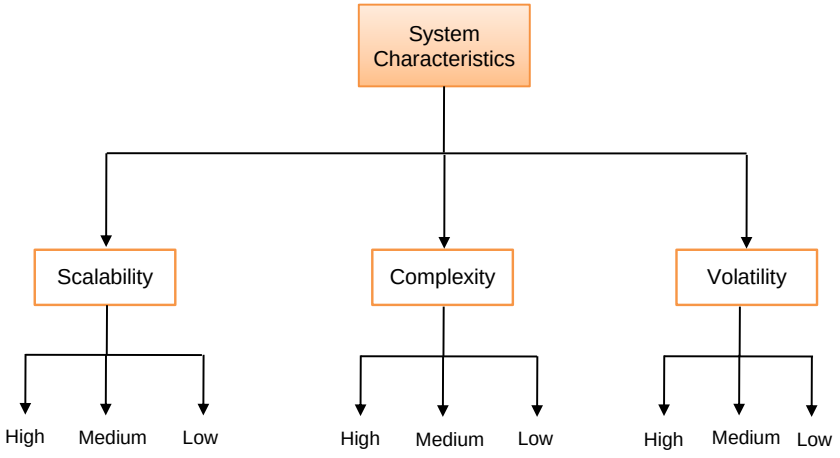


Figure 2.11: System characteristics taxonomy.

Complexity

According to the definition of Buyya et al. [23], a Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed autonomous resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements. The resources are heterogeneous and fault-prone as well as may be administered by different organizations. Thus Grid systems are complex by their characteristics.

However, as the complexity of a system increases, it becomes brittle and unmanageable. In that case, the system needs to be more autonomous so that it can handle the consequences of complexity without human intervention. For instance, the complexity of a decentralized Grid system is much higher than that of a centralized Grid system because of the interaction and coordination of the large number of decentralized components.

Scalability

Scalability of a distributed system is the ability for the system to easily expand or contract its resource pool to accommodate heavier or lighter loads. In other words, scalability is the ease with which a system can be modified, added or removed in order to accommodate varying workload.

If a system is highly scalable, its performance should not dramatically deteriorate as the system size increases. In general, decentralized or peer-to-peer Grid systems [99] are scalable in nature, whereas the centralized Grid systems [13] are least scalable as there exists a single point of control.

Volatility

Volatile refers to changing or changeable. Thus, volatility of a distributed system can be defined as the likelihood that the status of the system or its components to be altered due to the heterogeneous and dynamic behaviour of the environment, such as configuration change, resource failure, and load variation. If the condition of the system changes frequently over short period of time, it has high volatility. If the system status almost never changes, it has low volatility. In general, Grid systems are volatile in nature due to the

Table 2.3: Summary of Grid projects

Name	Organization	Status/ Availability	Application focus
Aneka Federation	The University of Melbourne, Australia http://www.gridbus.org	Free Evaluation version	Compute-intensive Bag-of-task
Askalon	University of Innsbruck, Austria http://dps.uibk.ac.at/askalon	Under Askalon Software License	Performance-oriented scientific Workflow
AutoMate	Rutgers University, USA http://www.caip.rutgers.edu/TASSL/Projects/AutoMate/	N/A	Data-intensive Bag-of-task
Condor-G	University of Wisconsin, USA http://www.cs.wisc.edu/condor/condorg/	Source code under Apache License	Compute-intensive Bag-of-task
GWMS	The University of Melbourne, Australia http://www.gridbus.org	Open source under GPL	Computational and data-intensive Workflow
Nimrod-G	Monash University, Australia http://messagelab.monash.edu.au/NimrodG	Source code under DSTC license	Computational and data-intensive Bag-of-task
Pegasus	University of Southern California, USA http://pegasus.isi.edu	Open source under Atlassian Confluence	Data-intensive Workflow
Taverna	Collaboration between several European institutes and industries http://taverna.sourceforge.net/	Source code under LGPL license	Bioinformatics Workflow
Triana	Cardiff University, UK http://www.trianacode.org/	Source code under Apache License	Compute-intensive Workflow

underlying characteristics.

2.6 Survey of Grid Systems

This section provides a detailed survey of selected existing Grid systems and mapping of the taxonomy proposed in the previous section onto these systems. Table 2.3 shows the summary of selected Grid workflow management projects. A comparison of various Grid systems and their categorization based on the taxonomy is shown in Table 2.4 to Table 2.9.

Table 2.4: Application composition taxonomy

Project	Application type	Application domain	Application definition	Data requirement
Aneka Federation	Bag-of-task	Business/Scientific	XML-based custom	Light
Askalon	Workflow	Scientific	AGWL-based custom	Light/Heavy
AutoMate	Bag-of-task	Business/Scientific	XML-based custom	Heavy
Condor-G	Bag-of-task/MPI	Scientific	ClassAd-based custom	Light
GWMS	Workflow	Scientific	xWFL-based custom	Heavy
Nimrod-G	Bag-of-task/MPI	Business/Scientific	DPML-based custom	Heavy
Pegasus	Workflow	Scientific	VDL-based custom	Heavy
Taverna	Workflow	Scientific	Scufl-based custom	Heavy
Triana	Workflow	Scientific	Tool-based	Light

Table 2.5: Application scheduling taxonomy

Project	Scheduling architecture	Scheduling objective	Scheduling decision	Scheduler integration
Aneka Federation	Decentralized	Load balancing	Dynamic	Combined
Askalon	Centralized	Utility/Optimization	Static/Dynamic	Separated/Combined
AutoMate	Decentralized	Load balancing	Dynamic	Combined
Condor-G	Centralized	Load balancing	Dynamic	Combined
GWMS	Centralized	Utility/Optimization	Dynamic	Separated
Nimrod-G	Centralized	Utility/Optimization	Dynamic	Separated/Combined
Pegasus	Centralized	Optimization	Static/Dynamic	Separated
Taverna	Centralized	Optimization	Dynamic	Separated/Combined
Triana	Centralized/Decentralized	Optimization	Dynamic	Separated/Combined

Table 2.6: Coordination taxonomy

Project	Decision making	Coordination Mechanism	Communication Protocol
Aneka Federation	Cooperative	Coordination space based	One-to-many
Askalon	Non-cooperative	Market/Group based	One-to-one
AutoMate	Cooperative	Coordination space based	One-to-many
Condor-G	Non-cooperative	Group based	One-to-one
GWMS	Non-cooperative	Market/Group based	One-to-one
Nimrod-G	Non-cooperative	Market/Group based	One-to-one
Pegasus	Non-cooperative	Group based	One-to-one
Taverna	Non-cooperative	Group based	One-to-one
Triana	Cooperative	Group based	One-to-many (all)

Table 2.7: Monitoring taxonomy

Project	Execution monitoring	Status monitoring	Directory Service
Aneka Federation	Passive	System health	Decentralized
Askalon	Passive	System health	Centralized
AutoMate	Passive	System health/Performance parameter	Decentralized
Condor-G	Active	System health	Centralized
GWMS	Passive	System health/Performance parameter	Centralized
Nimrod-G	Passive	System health/Performance parameter	Centralized
Pegasus	Active	System health	Centralized
Taverna	Passive	System health	Centralized
Triana	Passive	System health	Decentralized

Table 2.8: Self-* properties taxonomy

Project	Self-configuring	Self-optimizing	Self-healing	Self-protecting
Aneka Federation	Reconfiguration	Dynamic rescheduling	Failure detection	N.A.
Askalon	N.A.	Dynamic rescheduling/ Performance prediction	Failure detection/ Check-pointing	Authentication detection
AutoMate	Reconfiguration	Adaptive streaming	N.A.	Security policy
Condor-G	N.A.	Dynamic rescheduling	Failure detection	N.A.
GWMS	N.A.	Dynamic rescheduling	Failure detection	N.A.
Nimrod-G	N.A.	Dynamic rescheduling	Failure detection	N.A.
Pegasus	N.A.	Dynamic rescheduling	Failure detection/ Check-pointing	Authentication detection
Taverna	N.A.	Dynamic rescheduling	Failure detection	N.A.
Triana	Reconfiguration	Dynamic rescheduling	Failure detection/ Check-pointing	N.A.

Table 2.9: System characteristics taxonomy

Project	Scalability	Complexity	Volatility
Aneka Federation	High	High	High
Askalon	Medium	Medium	Medium
AutoMate	High	High	High
Condor-G	Low	Low	Medium
GWMS	Low	Medium	Medium
Nimrod-G	Low	Medium	Medium
Pegasus	Low	Medium	Medium
Taverna	Low	Medium	Medium
Triana	High	High	High

2.6.1 Aneka Federation

Aneka Federation system [99] logically connects topologically and administratively distributed Aneka Enterprise Grids as part of a single cooperative system. It uses a Distributed Hash Table (DHT), such as Pastry [107] and Chord [115] based self-configuring Peer-to-Peer (P2P) network model for discovering and coordinating the provisioning of distributed resources in Aneka Grids. It also employs a novel resource provisioning technique that assigns the best possible resource sets for the execution of applications, based on their current utilization and availability in the system.

The application scheduling and resource discovery in Aneka-Federation is facilitated by a specialized Grid Resource Management System, known as Aneka Coordinator (AC). AC is composed of three software entities: Grid Resource Manager (GRM), Local Resource Management System (LRMS), and Grid Peer. The GRM component of AC exports a Grid site to the federation and is responsible for coordinating federation wide application scheduling and resource allocation. GRM is also responsible for scheduling locally submitted jobs in the federation using LRMS.

Grid peer implements a DHT based P2P overlay for enabling decentralized and distributed resource discovery supporting resources status lookups and updates across the federation. It also enables decentralized inter-AC collaboration for optimizing load-balancing and distributed resource provisioning. The employment of DHT improves system scalability by enabling the ability to perform deterministic discovery of resources and produces controllable number of messages (by using selective broadcast approach) in comparison to using other One-to-All broadcast techniques, such as JXTA [54] that is a set of protocols for decentralized P2P applications.

Distributed trust mechanism is utilized in Aneka Federation to ensure secured resource management across the federation. Furthermore, the Aneka Container component of AC provides the base infrastructure that consists of services for persistence and security (authorization, authentication and auditing).

2.6.2 Askalon

Askalon [39] is a Grid application development and execution environment. The goal of Askalon is to simplify the development and optimization of scientific workflow applications that can harness the power of computational Grids (i.e. Austrian Grid). Askalon is comprised of four tools (Scalea, Zenturio, Aksum, and PerformanceProphet), coherently integrated into a service-oriented architecture. Scalea is a performance measurement and analysis tool for parallel and distributed high performance applications. Zenturio is a general purpose experiment management tool with the support for multi-experiment performance analysis and parameter studies. Through a special-purpose performance property specification language, Aksum provides semi-automatic high level performance bottleneck detection. The PerformanceProphet facilitates the users in terms of modeling and predicting the performance of parallel applications at the early stages of development.

Askalon uses the XML-based Abstract Grid Workflow Language (AGWL) [40] for composing workflow applications. The Scheduler service processes the workflow specification described in AGWL, converts it to executable form and maps it onto the available Grid resources. Resource Manager is utilized to retrieve the current status and availability of Grid resources. Furthermore, the Enactment Engine coordinates the execution of workflow tasks according to the control flows and data dependencies specified by the application developers.

Askalon employs a hybrid approach for scheduling workflow applications on the Grid through dynamic monitoring and steering, combined with static optimization. Static optimization maps entire workflow onto the Grid resources using Genetic Algorithm based on user-defined Quality of Service (QoS) parameters. A dynamic scheduling algorithm then takes into consideration the dynamic nature of the Grid resources, such as machine crashes or external CPU and network load. Askalon also employs self-healing mechanism through checkpointing and migration techniques that support reliable workflow execution in presence of resource failures as well as when the Enactment Engine itself crashes.

In order to establish authentication mechanism across Askalon user portals and Grid services, Grid Security Infrastructure (GSI) [44] has been employed based on single sign-on, credential delegation, and web services security (through XML digital signature and

XML encryption).

2.6.3 AutoMate

The main objective of AutoMate [91] is to develop conceptual models and implementation architectures that enable the development and execution of self-managing Grid applications. The major components of AutoMate are: Accord programming framework, Rudder coordination middleware, Meteor content-based middleware and Sesame access control engine. In AutoMate, application composition plan is generated by Accord, element discovery is performed by Meteor and plan execution is achieved by Rudder. AutoMate portals provide users with secure, pervasive and collaborative access to different components and entities.

The Accord programming framework extends existing distributed programming models and frameworks to addresses the definition, execution and runtime management of autonomic elements. In particular, it extends the entities and composition rules defined by the underlying programming model to enable computational and composition/interaction behaviors to be defined at runtime using high level rules. In Accord, composition plans are generated using the Accord Composition Engine and are expressed in XML.

The Rudder coordination middleware provides the core capabilities for supporting autonomic composition, adaptation, and optimization. It builds upon two concepts: Context-aware agent framework and Decentralized tuple space. A context-aware agent is a processing unit that performs tasks to automate the control and coordination of the autonomic elements. The decentralized tuple space scalably and reliably supports the distributed agent-based system coordination.

The Meteor content-based middleware provides support for content-based routing, decentralized information discovery and messaging service through Squid and Pawn. Squid is a Distributed Hash Table (DHT) [106] based P2P system that enables efficient and scalable information discovery, while supporting complex queries. Pawn is a P2P messaging substrate that builds on JXTA [54] to support P2P interactions in the Grid. Pawn provides a stateful and guaranteed messaging to enable key application-level interactions, such as synchronous/asynchronous communication, dynamic data injection, and remote

procedure calls.

In AutoMate, secure interaction among the participating entities is managed by AutoMate Access Control Engine, Sesame. It is composed of access control agents and provides dynamic role based access control to users, applications, services, and resources.

2.6.4 Condor-G

Condor-G [47] is the combination of technologies from the Condor project and the Globus project [42]. It combines the inter-domain resource management protocols of the Globus Toolkit and the intra-domain resource and job management mechanisms of Condor. In particular, Condor-G leverages security and resource access in multi-domain environments, as supported by the Globus Toolkit as well as management of computation and harnessing of resources within a single administrative domain, embodied by the Condor system. Its flexible and intuitive commands are appropriate for use directly by end-users, or for interfacing with higher-level task brokers and web portals.

The computation management service of Condor-G is called Condor-G agent. It allows users to treat the Grid as an entirely local resource, with an API and command line tools that facilitate them to perform several job management operations: (1) submit jobs by indicating an executable name, input/output files, and arguments; (2) query a job's status or cancel the job; (3) be informed of job termination/problems via callbacks or asynchronous mechanisms, such as email; (4) obtain access to detailed log that provides a complete history of the jobs' execution.

Condor-G comprises of a powerful, full-featured task broker/scheduler that can manage thousands of jobs destined to run at distributed sites. It supports job scheduling, monitoring, policy enforcement, fault tolerance, and credential management and handles complex job-interdependencies. Specifically, the job-interdependencies are handled by the associated meta-scheduler, Directed Acyclic Graph Manager (DAGMan) [2]. While Condor-G aims to discover available machines for the execution of jobs, DAGMan handles the dependencies between the jobs. The resource brokering is done through a matchmaking algorithm.

The Condor-G scheduler responds to a user request of submitting jobs to be run on

Grids by creating a new Condor-G GridManager daemon. One GridManager process handles all jobs for a single user and terminates once all jobs are completed. The job submission request of each GridManager results in the creation of one Globus JobManager [42] daemon. The GridManager detects remote failures by periodically probing the JobManagers of all the jobs it manages and resubmits the failed jobs once detected.

2.6.5 GWMS

Gridbus Workflow Management System (GWMS) [134] facilitates users to execute their workflow applications on Grids. The two main components of GWMS are workflow portal and workflow engine. The primary user interface for the users to access GWMS is a web portal that comprises an editor and a monitor component. The workflow editor provides a Graphical User Interface (GUI) and allows users to create new and modify existing workflows utilizing the drag and drop facilities. Workflow monitor provides a GUI for viewing the status of each task in the workflow. Users can also view the site of execution for each task, the number of tasks being executed, and the failure history of each task.

Workflow engine is designed to support an XML-based WorkFlow Language (xWFL). This facilitates user level planning at the submission time. The workflow language parser converts workflow description from XML format to Tasks, Parameters and Data Constraints (workflow dependency), which are accessed by workflow scheduler. The resource discovery component of the engine sends query to Grid Information Services, such as Globus MDS [41], directory service, and replica catalogues to locate suitable resources for execution of the tasks in the workflow. It also uses Gridbus Broker [126] for dispatching and managing task executions on different Grid sites comprising various middlewares. Execution of workflow tasks on different Grid middlewares is achieved by the creation of specific dispatchers for corresponding middleware.

Workflow engine also employs a just-in-time scheduling system using tuple space, where every task has its own scheduler called Task Manager (TM), which implements a scheduling algorithm and handles the processing of tasks. The TMs are controlled by a Workflow Coordinator. Although these TMs work in a distributed fashion, they

communicate with each other through the tuple space that is designed based on a client-server based centralized technology. However, the just-in-time scheduling system allows resource allocation decision to be made at the time of task execution, and hence adapts to the changing Grid environments. Task failures are handled in GWMS by resubmitting the failed tasks to resources that do not have failure history for these tasks.

2.6.6 Nimrod-G

Nimrod/G [21] is a widely adopted Grid middleware environment for building and managing large computational experiments over distributed resources. It uses the Globus [42] middleware services for dynamic resource discovery and job dispatching in computational Grids. The main components of Nimrod/G are: Client or User Station, Parametric Engine, Scheduler, Dispatcher, and Job-Wrapper. In addition, it provides a web based interface that allows users to create and manage experiments without installing Nimrod/G client locally.

Client or User Station acts as a user-interface for controlling and supervising an experiment under consideration. The user can vary parameters related to execution time and cost, which influence the scheduling decision while selecting resources. It also serves as a monitoring console for users and lists status of all jobs.

The Parametric Engine is the core component of Nimrod/G and acts as a persistent job control agent that handles the whole experiment. It is responsible for parameterization of the experiment, creation of jobs, maintenance of job status, and interaction with Nimrod scheduler and clients. The engine takes the experiment plan, described using a Declarative Parametric Modeling Language (DPML) as input and manages the experiment under the direction of scheduler.

The Scheduler is responsible for resource discovery, resource selection, and job assignment. The resource discovery process interacts with a Grid information service directory (MDS in Globus), identifies the list of authorized machines, and keeps track of resource status information. As Nimrod/G incorporates computational economy based job scheduling approach, the resource selection process selects the resources that meet the execution completion deadline set by the user as well as minimize the cost of compu-

tation.

The Dispatcher primarily initiates the execution of tasks in a job on the selected resources according to scheduler's instruction. The Job-wrapper is responsible for staging the application data, starting execution of the tasks on assigned resources, and sending results back to the Parametric Engine through Dispatcher. The architecture of Nimrod/G is extensible enough to support job execution in several Grid middleware services, such as Legion [27] and Condor [117] by implementing specific job dispatcher for the corresponding middleware.

2.6.7 Pegasus

Pegasus Workflow Management System (PWMS) [37] has been developed as part of the GriPhyN project [5] that aims to support large-scale data management in physics experiments, such as high-energy physics and astronomy. It maps and executes complex scientific workflows on the Grid. PWMS is composed of two major components: Pegasus workflow mapping engine and DAGMan workflow executor for Condor.

Pegasus workflow mapping engine receives an abstract workflow description expressed in Chimera's [46] Virtual Data Language (VDL) and generates an optimized concrete workflow by mapping workflow tasks to a set of available Grid resources. The abstract workflow describes the tasks and data in terms of their logical names and indicates their dependencies in the form of DAG, whereas concrete workflow specifies the location of the data and the task execution platforms. Pegasus uses the centralized meta-scheduler, DAGMan [2] as enactment engine with the enhancement of data derivation techniques that simplify the workflow at run-time based on data availability. Thus, combined with DAGMan, Pegasus is able to map and execute workflows on a variety of platforms, such as Condor pools, Cluster managed by LSF or PBS, TeraGrid hosts and individual hosts.

In order to locate the replicas of the required data and find the location of logical application components, Pegasus uses Replica Location Service (RLS) and Transformation Catalog (TC) respectively. It also queries Globus Monitoring and Discovery Service (MDS) to find out available resources and their characteristics. Recently Pegasus is integrated with a workflow creation system named Wings [52], which utilizes semantic

representations to manage workflows that process large data and computation steps [83]. These are represented using a subset of OWL-DL (description logic-based language) in the form of file and component ontologies.

Pegasus uses two methods for resource selection: random allocation and performance prediction. In the later approach, Pegasus interacts with Prophesy [129] that is used to predict the best site to execute an application component by using performance historical data. It adopts the strategy of dynamically adjusting resource allocation decisions in response to feedback on the performance of workflow execution. This adaptive strategy is structured around the MAPE functional decomposition, which partitions the adaptive functionalities into four areas: Monitoring, Analysis, Planning, and Execution. In case of job failure, Pegasus incorporates Retry policy and generates a rescue DAG by DAGMan, which is modified and resubmitted at a later time.

2.6.8 Taverna

Taverna [85] is a workflow management tool of the myGrid project [114], which aims to exploit Grid technology to develop high-level middleware for supporting data-intensive in silico bioinformatics experiments using distributed resources. The tool includes a workbench application, called Scufl Workbench that provides a graphical user interface for the composition of workflows and an enactment engine, called Freefluo enactor that facilitates transfer of intermediate data and invocation of web services.

In Taverna, a workflow is considered as a graph of processors, each of which transforms a set of data inputs into a set of data outputs. These workflows are written in a language, called Simple Conceptual Unified Flow Language (SCUFL), where each processor within a workflow represents one atomic task.

The Scufl Workbench enables bioinformaticians to compose workflows without having to learn SCUFL. It acts as a container for a number of user interface components and provides a user-friendly multi-window environment for users to manipulate workflows, validate and select available resources as well as execute and monitor these workflows. Scufl language parser is used to parse Scufl workflow definitions into a form that is enacted by the Freefluo enactor.

Workflows are executed on the Scufl workbench using the enactor launch panel. This panel allows inputs to be specified for the workflow and launches a local instance of the Freefluo enactment engine. Freefluo is a Java based workflow orchestration tool. It supports invoking different types of services, such as WSDL-based [7] single operation web services, Soaplab bio-services, Talisman and local applications. The enactment status panel of Taverna shows the current progress of a workflow invocation and allows users to browse the intermediate and final results.

Fault tolerance in Taverna is achieved by setting configuration (e.g. number of retries, time delay, and alternative processor) for each processor in the workflow. It also allows users to specify the critical level for faults on each processor. If a processor is set as Critical, when all retries and alternatives are failed, entire workflow execution is terminated; otherwise the execution of workflow is continued, but children nodes of the failed processor are never invoked.

2.6.9 Triana

Triana [118] is a workflow composition and management environment that consists of an intuitive Graphical User Interface (GUI) for application composition and an underlying subsystem, which allows integration of Triana with multiple Grid services and interfaces. The GUI consists of two main components: Tool browser and Work surface. Tool browser employs a conventional file browser interface and Work surface is used to graphically connect the tools to form a dataflow diagram. Thus, users create applications by dragging the desired tools (or services) from the Tool browser onto the Work surface, and then wiring them together to create a workflow or dataflow for specific behavior.

The underlying subsystem consists of a collection of interfaces, consisting of various middlewares and services, including the Grid Application Toolkit (GAT) that integrates Triana into the Grid. GAT defines a high level API for access to core Grid services using JXTA [54], web services, and OGSA (Open Grid Services Architecture). Triana supports two types of application components for distributed execution: Grid-oriented and Service-oriented. Grid-oriented components refer to applications that are executed on the Grid using Grid resource manager, such as Grid Resource Management System (GRMS).

Service-oriented components are remote applications that are invoked through network interfaces, such as Web services and JXTA services.

An important feature of Triana is that it enables users to distribute sections of a workflow to remote machines for execution. The distribution of workflow requires the existence of Triana launcher services running on the remote machines. A launcher service provides the contact point for Triana on the remote machine and facilitates the creation of actual Triana services executing workflow subsections. Moreover, Triana connects input and output pipes to nodes of the remote service for enabling the data to be passed from the local workflow to the remote service, and the results to be passed back to the user.

The distribution mechanism is facilitated by the GAT interface that is not bound to any specific middleware. Currently, Triana workflow environment supports two types of GAT bindings, one for JXTA and another for P2PS, a simple socket based P2P toolkit.

2.7 Thesis Scope and Positioning

From the taxonomy and survey, we identify that most of the existing Grid workflow management systems in Grids are centralized and do not support cooperative application scheduling. In addition, as these Grid systems are highly complex and volatile, most of them incorporate self-optimizing and self-healing properties of an autonomic computing system. However, in order to cope up with the increasing-scale complexity and volatility of Grid environment, these systems are also required to address the self-configuring and self-protecting policies to some extent. In the thesis, we have limited focus on enabling the self-protecting property. Nevertheless, we address the other self-* properties, such as self-configuring, self-optimizing, and self-healing in our proposed autonomic workflow management system. We also propose a decentralized coordination space based coordination policy for enabling cooperative scheduling of scientific workflow applications as well as utilize active monitoring technique for continuous adaptation and optimization with regards to the dynamic Grid environment. The scope of this thesis with respect to the autonomic computing features covered within the taxonomy is presented in Table 2.10.

Table 2.10: Positioning of thesis with respect to taxonomy

Features of AC	**Scope of thesis**
Application Composition	scientific workflows
Application Scheduling	decentralized, dynamic, and optimization/load balancing
Coordination	Coordination space based and cooperative
Monitoring	active, decentralized, and system health
Self-* Property	self-configuring, self-optimizing, and self-healing
System Characteristics	highly scalable, complex, and volatile

2.8 Conclusion

This chapter presents an overview and state-of-the-art of Autonomic Application Management (AAM) in Grid computing environment. After analyzing the AAM landscape, we categorize the process of facilitating AAM according to various aspects of application management and propose a comprehensive taxonomy based on six different perspectives: (i) application composition, (ii) application scheduling, (iii) coordination, (iv) monitoring, (v) self-* property, and (vi) system characteristics. Further, we develop taxonomies for each of these perspectives to classify the common trends, solutions, and techniques in application management. Hereby, we provide pointers to related research work in this context. Next, a survey is also conducted, where the taxonomy is mapped to the selected Grid systems mostly focused on scientific workflow management. The survey helps to analyze the gap between what autonomic application management policies and methodologies are already available in existing Grid systems and what are still required to be addressed so that some outstanding research issues can be identified.

This chapter also discusses positioning of the thesis with respect to the proposed taxonomy in terms of AAM. The main research direction of this thesis in relation to the taxonomy is focused on developing self-optimizing and self-healing scheduling strategy for managing workflow applications in a cooperative, but complex and volatile Grid environment. The next chapters describe in detail our contributions towards this research direction, starting with understanding the concepts and methodologies for workflow scheduling in dynamic and heterogeneous Grid computing environment.

Chapter 3

DCP-G: A Dynamic Workflow Scheduling Algorithm for Grids

Effective scheduling is a key concern for the execution of performance driven Grid applications such as workflows. In this chapter, we first define the workflow scheduling problem and describe the existing heuristic and meta-heuristic based workflow scheduling strategies in Grids. Then we propose a dynamic critical path based workflow scheduling algorithm for Grids, which determines efficient mapping of workflow tasks to Grid resources by calculating the critical path in the workflow task graph at every step. Using simulation, the performance of the proposed approach is compared with the existing approaches, discussed in this chapter for different type and size of workflows. The results demonstrate that the heuristic based scheduling techniques can adapt to the dynamic nature of resource and avoid performance degradation in dynamically changing Grid environments.

3.1 Introduction

Many of the large-scale scientific applications executed on present-day Grids are expressed as complex e-Science workflow [73] [18], which is a set of ordered tasks that are linked by data dependencies. A Workflow Management System (WMS) [133] is generally employed to define, manage and execute these workflow applications on Grid resources.

A WMS uses a specific scheduling strategy for mapping the tasks in a workflow to suitable Grid resources in order to satisfy user requirements. Numerous workflow scheduling strategies have been proposed in literature for different objective functions [137].

However, the majority of these scheduling strategies are static in nature. They produce a good schedule given the current state of Grid resources and do not take into account changes in resource availability. On the other hand, dynamic scheduling is done on-the-fly considering the current state of the system and adaptive in nature. Thus, in this chapter, we present a dynamic workflow scheduling technique that not only dynamically minimizes the workflow execution time but also reduces the scheduling overhead constituting a significant amount of time used by scheduler to generate schedule.

Critical path heuristics [71] have been used extensively for scheduling interdependent tasks in multiprocessor systems. These heuristics aim to determine the longest of all execution paths from the beginning to the end (or the *critical path*) in a task graph, and schedule them earliest so as to minimize the execution time for the entire graph. Kwok and Ahmad [72] introduced the Dynamic Critical Path (DCP) algorithm in which the critical path is dynamically determined after each task is scheduled. However, this algorithm is designed for mapping tasks on to homogeneous processors, and is static, in the sense that the schedule is only computed once for a task graph. We extend the DCP algorithm to map and schedule tasks in a workflow on to heterogeneous resources in a dynamic Grid environment. In order to evaluate the performance of our proposed algorithm, called DCP-G (Dynamic Critical Path for Grids), we have compared it against the existing approaches, discussed in this chapter for different types and sizes of workflows. The results demonstrate that DCP-G can adapt to temporal resource behavior and avoid performance degradation in dynamically changing Grid environments.

3.2 Background of Workflow Scheduling

3.2.1 Workflow Scheduling Problem

In General, a workflow application is represented as a Directed Acyclic Graph (DAG) in which graph nodes represent tasks and graph edges represent data dependencies among

the tasks with weights on the nodes representing computation complexity and weights on the edges representing communication volume. Therefore, workflow scheduling problem is usually considered as a special case of the DAG scheduling problem, which is in an NP-complete problem [50]. Thus, even though the DAG scheduling problem can be solved by using exhaustive search methods, the complexity of generating the schedule becomes very high.

The overall finish/completion time of an application is usually called the schedule length or makespan. So the objective of workflow scheduling techniques is to minimize the makespan of a parallel application by proper allocation of the tasks to the processors/resources and arrangement of task execution sequences.

Let us assume, workflow $W(T, E)$ consists of a set of tasks, $T = \{T_1, T_2, \ldots, T_x, .. , T_y, T_n\}$ and a set of dependencies among the tasks, $E = \{< T_a, T_b >, ..., < T_x, T_y >\}$, where T_x is the parent task of T_y. $R = \{R_1, R_2, \ldots, R_x, \ldots, R_y, R_m\}$ is the set of available resources in the computational Grid. Therefore, the workflow scheduling problem is the mapping of workflow tasks to Grid resources ($T \rightarrow R$) so that the makespan, M is minimized.

Generally, a workflow task is a set of instructions that can be executed on a single processing element of a computing resource. In a workflow, an entry task does not have any parent task and an exit task does not have any child task. In addition, a child task can not be executed until all of its parent tasks are completed. At any time of scheduling, the task that has all of its parent tasks finished, is called a ready task.

3.2.2 Existing Workflow Scheduling Algorithms

As workflow scheduling is an NP-complete problem, we rely on heuristic and meta-heuristic based scheduling strategies to achieve near optimal solutions within polynomial time. In the following, we present some of the well known heuristics and meta-heuristics for workflow scheduling in Grid systems.

3.2.3 Heuristics

Myopic

Myopic is an individual task scheduling heuristic that is considered as the simplest scheduling method for scheduling workflow applications because it makes scheduling decisions based on only one individual task. The Myopic algorithm presented in [128] has been implemented in some Grid systems such as Condor DAGMan [2]. It schedules an unmapped ready task, in arbitrary order to the resource, which is expected to complete that task earliest, until all tasks have been scheduled.

Min-Min

A list scheduling heuristic prioritizes workflow tasks and schedules the tasks based on their priorities. Min-Min is a list scheduling heuristic that assigns the task priority based on its Expected Completion Time (ECT) on a resource. This heuristic organizes the workflow tasks into several independent task groups and schedules each group of independent tasks iteratively. In every iteration, it takes the set of all unmapped independent tasks T and generates the Minimum Expected Completion Times (MCT) for each task t in T.

where $MCT_t = min_{r \epsilon R} ECT(t, r)$; R is the set of resources available; and $ECT(t, r)$ is the amount of time resource r takes to execute task t.

Then, the task having minimum MCT value over all tasks is selected to be scheduled first at this iteration to the corresponding resource for this MCT (hence the name is Min-Min). In this way, Min-Min schedules other independent tasks in T and moves to next iteration until T is empty.

The intuition behind Min-Min is to consider all unmapped independent tasks during each mapping decision, whereas Myopic only considers one task at a time. Min-Min was proposed by Maheswaran et al. [80] and has been employed for scheduling workflow tasks in Grid projects such as vGrADS [19] and Pegasus [81].

Max-Min

The Max-Min heuristic is very similar to Min-Min. The only difference is the Max-Min heuristic sets the priority to the task that requires the longest execution time rather than

Table 3.1: Summary of workflow scheduling algorithms

Scheduling method	Scheduling type	Project	Organization
Myopic	Heuristic	Condor DAGMan	University of Wisconsin-Madesion, USA
Min-Min	Heuristic	vGrADS	Rice University, USA
Max-Min	Heuristic	vGrADS	Rice University, USA
HEFT	Heuristic	ASKALON	University of Innsbruck, Austria
GRASP	Meta-heuristic	Pegasus	University of Southern California
GA	Meta-heuristic	ASKALON	University of Innsbruck, Austria

the shortest execution time. In each iterative step, after obtaining the set of MCT values for all unmapped independent tasks, a task having the maximum MCT is chosen to be scheduled on the resource, which is expected to complete the task at the earliest time.

Intuitively, Max-Min attempts to minimize the total workflow execution time by assigning longer tasks to comparatively best resources. Max-Min was also proposed by Maheswaran et al. [80] and has been used for scheduling workflow tasks in Pegasus [81].

HEFT

Heterogeneous Earliest Finish Time (HEFT) [121] is a well established list scheduling algorithm, which gives higher priority to the workflow task having higher rank value. This rank value is calculated by utilizing average execution time for each task and average communication time between resources of two successive tasks, where the tasks in Critical Path get comparatively higher rank values. Then it sorts the tasks by decreasing order of their rank values and the task with higher rank value is given higher priority. In the resource selection phase, tasks are scheduled in the order of their priorities and each task is assigned to the resource that can complete the task at the earliest time.

The advantage of using this technique over Min-Min or Max-Min is that while assigning priorities to the tasks it considers the entire workflow rather than focusing on only unmapped independent tasks at each step. HEFT algorithm was proposed by Topcuoglu et al. [121] and it has been used in the ASKALON workflow manager [128] [39] to provide scheduling for a quantum chemistry application WIEN2K [18].

3.2.4 Meta-heuristics

GRASP

Greedy Randomized Adaptive Search Procedure (GRASP) [19] is an iterative randomized search technique. In GRASP, a number of iterations are conducted to search a possible optimal solution for mapping tasks on resources. A solution is generated at each iterative step and the best solution is kept as the final schedule. This searching procedure terminates when the specified termination criterion, such as the completion of a certain number of iterations, is satisfied. GRASP can generate better schedules than the other scheduling techniques stated previously as it searches the whole solution space considering entire workflow and available resources.

GA

Likewise GRASP, Genetic Algorithm (GA) [53] is also a meta-heuristic based scheduling technique that allows a high quality solution to be derived from a large search space in polynomial time by applying the principles of evolution. A genetic algorithm combines exploitation of best solution from past searches with the exploration of new regions of solution space. Instead of creating a new solution by randomized search as in GRASP, GA generates new solutions at each step by randomly modifying the good solutions generated in previous steps, which results in a better schedule within less time. Jia et al. [135] have employed GA based approach to schedule workflows in Grids.

3.3 DCP-G Algorithm for Workflow Scheduling

For a task graph, the lower and upper bounds of starting time for a task are denoted as the Absolute Earliest Start Time (AEST) and the Absolute Latest Start Time (ALST) respectively. In the DCP algorithm [72], the tasks on the critical path have equal AEST and ALST values as delaying these tasks affects the overall execution time for the task graph. The first task on the critical path is mapped to the processor identified for it. This process is repeated until all the tasks in the graph are mapped.

However, this algorithm is designed for scheduling all the tasks in a task graph with

Table 3.2: Symbols and Their Meanings

Symbol	Meaning
$AET(t)$	Absolute execution time of task t
$ADTT(t)$	Absolute data transfer time for task t
$AEST(t, R)$	Absolute earliest start time of task t on resource R
$ALST(t, R)$	Absolute latest start time of task t on resource R
$C_{t,t_k}(R_t, R_{t_k})$	Data transfer time between task t and t_k that are scheduled to resources R_t and R_{t_k} respectively
$PC(R)$	Processing capacity of resource R
$BW(R)$	Bandwidth of the network link that connects resource R to Global Grid
$DCPL$	Length of a dynamic critical path in a workflow

arbitrary computation and communication times to a multiprocessor system with unlimited number of fully connected identical processors. But, Grids [43] are heterogeneous and dynamic environments consisting of computing, storage and network resources with different capabilities and availability. Therefore, to work on Grids, the DCP algorithm needs to be extended in the following manner:

- For a task, the initial AEST and ALST values are calculated for the resource, which provides the minimum execution time for the task. The overall objective is to reduce the length of the critical path at every pass. We follow the intuition of the Min-Min heuristic in which a task is assigned to the resource that executes it fastest.

- For mapping a task on the critical path, all the available resources are considered by DCP-G, as opposed to the DCP algorithm, which considers only the resources (processors) occupied by the parent and child tasks. This is because, in the latter case, the execution time is not varied for different processors, and only the communication time between the tasks could be reduced by assigning tasks to the same resource. However, in Grids, the communication and computation times are both liable to change because of resource heterogeneity.

- When a task is mapped to a resource, its execution time and data transfer time from the parent node are updated accordingly. This changes the AEST and ALST of succeeding tasks.

In the following, we discuss some of the principle features of the algorithm. In the first part of the discussion, we describe the techniques used to calculate AEST and ALST that

are necessary for task selection. Then we discuss the task selection methodology followed by the resource selection strategy. The DCP-G scheduling algorithm is formalized and illustrated with an example at the end of this section. Table 3.2 provides some terms and their meanings that are used in the subsequent discussion.

3.3.1 Calculation of AEST and ALST in DCP-G

In DCP-G, the start time of a task is not finalized until it is mapped to a resource. Here, we also introduce two more attributes: the Absolute Execution Time (AET) of a task, which is the minimum execution time of the task, and Absolute Data Transfer Time (ADTT), which is the minimum time required to transfer the output of the task given its current placement. Initially, AET and ADTT are calculated as,

$$AET(t) = \frac{Task\_size(t)}{MAX_{k \epsilon ResourceList}\{PC(R_k)\}}$$

$$ADTT(t) = \frac{Task\_output\_size(t)}{MAX_{k \epsilon ResourceList}\{BW(R_k)\}}$$

where $PC(R_k)$ and $BW(R_k)$ are processing capability and transfer capacity (i.e. bandwidth) of resource R_k respectively.

Whenever a task t is scheduled to a resource, the values of $AET(t)$ and $ADTT(t)$ are updated accordingly. Therefore, the AEST of a task t on resource R, denoted by $AEST(t, R)$ is recursively defined as:

$$AEST(t, R) = MAX_{1 \leq k \leq p}\{AEST(t_k, R_{t_k}) + AET(t_k) + C_{t,t_k}(R_t, R_{t_k})\}$$

where t has p parent tasks, t_k is the k^{th} parent task and,
$AEST(t, R) = 0$; if t is an entry task.
$C_{t,t_k}(R_t, R_{t_k}) = 0$; if $R_t = R_{t_k}$.
$C_{t,t_k}(R_t, R_{t_k}) = ADTT(t_k)$; if t and t_k are not scheduled.

Here, the communication time between two tasks is considered to be zero if they are

mapped to the same resource and equal to the ADTT of the parent task if the child is not mapped yet. Using this definition, the AEST values can be computed by traversing the task graph in a breadth-first manner beginning from the entry tasks.

Once AESTs of all the tasks are computed, it is possible to calculate Dynamic Critical Path Length (DCPL), which is the schedule length of the partially mapped workflow. DCPL can be defined as:

$$DCPL = MAX_{1 \leq i \leq n}\{AEST(t_i, R_{t_i}) + AET(t_i)\}$$

where n is the total number of tasks in the workflow.

After computing the DCPL, the values of ALST can be calculated by traversing the task graph in a breadth first manner but in the reverse direction. Thus, the ALST of a task t in resource R, denoted as ALST(t,R) can be recursively defined as,

$$ALST(t, R) = MIN_{1 \leq k \leq c}\{ALST(t_k, R_{t_k}) - AET(t) - C_{t,t_k}(R_t, R_{t_k})\}$$

where t has c child tasks, t_k is the k^{th} child task and

$ALST(t, R) = DCPL - AET(t)$; if t is an exit task.

$C_{t,t_k}(R_t, R_{t_k}) = 0$; if $R_t = R_{t_k}$.

$C_{t,t_k}(R_t, R_{t_k}) = ADTT(t_k)$; if t and t_k are not mapped.

3.3.2 Task Selection

During the scheduling process, the Critical Path (CP) in the task graph determines the schedule length of the partially scheduled workflow. Thus, while scheduling, it is necessary to give priority to the tasks in CP. However, as the scheduling process progresses, the CP can be changed dynamically i.e. a task on a CP at one step may not be on CP at the next step because of the dynamically changing resource behaviour. This is why, in dynamic environment such as Grids, CP in the workflow is called Dynamic Critical Path since it is likely to be changed at every step of scheduling.

The tasks on DCP have the same upper and lower bound of start time. Therefore, a task in DCP-G is considered to be on the critical path if its AEST and ALST values are equal. In order to reduce the value of DCPL at every step, the task selected for scheduling is the one that is on CP and has no unmapped parent tasks.

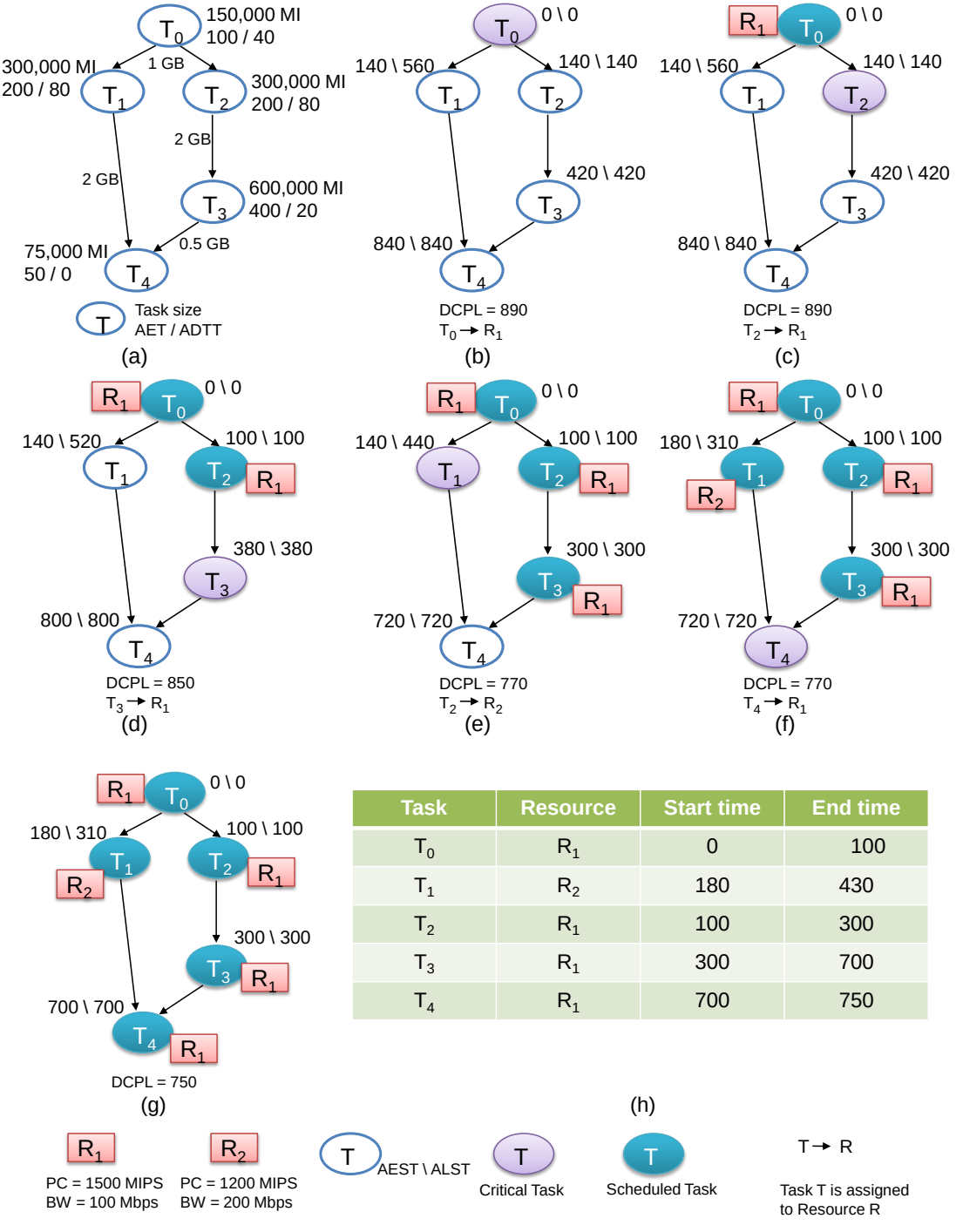
3.3.3 Resource Selection

After identifying a critical task, we need to select an appropriate resource for that task. We select the resource that provides the minimum execution time for that task. This is discovered by checking all the available resources for one that minimizes the potential start time of the critical child task on the same resource, where the critical child task is the one with the least difference of AEST and ALST among all the child tasks of the critical task. Finally, the critical task is mapped to the resource that provides the earliest combined start time.

3.3.4 Methodology

First, the DCP-G algorithm (refer to Algorithm 1) computes initial AET, ADTT, AEST and ALST of all the tasks. Then it selects the task with the smallest difference between its AEST and ALST, where ties are broken by choosing the one with smaller AEST. According to the discussion in Section 3.3.2, this task is on DCP and called Critical Task (CT). The critical child task of CT is also determined in the same manner. The algorithm then computes the start time of CT for all available resources considering the finish time of all of its parent tasks and searches for a slot starting with this start time with the duration of its execution time. The resource that gives the earliest start time for both t_{ct}, and its critical child task is selected.

After selecting the suitable resource R, the algorithm calculates start time $AEST(t_{ct}, R)$ and duration $AET(t_{ct})$ for t_{ct} on this resource, and updates the actual start and execution times for t_{ct} accordingly. The AEST and ALST values of other tasks are updated at the end of each scheduling step in order to determine next critical task. This process continues until all the tasks in the workflow are scheduled.



Task	Resource	Start time	End time
T_0	R_1	0	100
T_1	R_2	180	430
T_2	R_1	100	300
T_3	R_1	300	700
T_4	R_1	700	750

Figure 3.1: Example of workflow scheduling using DCP-G algorithm.

Algorithm 1 DCP scheduling algorithm

PROCEDURE: DCP
Input: Workflow $W(T, E)$, $TaskDependencyList$, Resource Set R
begin
  for all $t \in T$ of workflow W
    Calculate AET and ADTT for t according to Section 3.3.1
  end for
  for all $t \in T$ of workflow W
    Calculate AEST for t running BFS according to Section 3.3.1
  end for
  Calculate DCPL
  for all $t \in T$ of workflow W
    Calculate ALST for t running BFS following reverse task dependency according to Section 3.3.1
  end for
  while all tasks in T are not completed **do**
    $TaskList \leftarrow$ Get unscheduled $Ready$ tasks from workflow W
    Schedule Task ($TaskList$, R)
    Update $TaskDependencyList$
  end while
end
PROCEDURE: Schedule Task
Input: $TaskList$, Resource Set R
begin
  while $TaskList$ is **not** empty **do**
    $T_{ct} \leftarrow$ Get Critical Task from all tasks of $TaskList$ according to Section 3.3.2
    $T_{ct_c} \leftarrow$ Get Critical Child Task from all child tasks of T_{ct} according to Section 3.3.3
    $r \leftarrow$ Get a resource from R that can provide earliest start time for both T_{ct} and T_{ct_c}
    Schedule T_{ct} on r
    Update status of r
    Remove T_{ct} from $TaskList$
    for all $t \in T$ of workflow W
      Calculate AEST for t running BFS
    end for
    Calculate DCPL
    for all $t \in T$ of workflow W
      Calculate ALST for t running BFS following reverse task dependency
    end for
  end while
end

3.3.5 DCP-G Example

Fig. 3.1 illustrates the DCP-G algorithm with a step-by-step explanation of the mapping of tasks in a sample workflow. The sample workflow consists of five tasks denoted as T_0, T_1, T_2, T_3, and T_4 with different execution and data transfer requirements. The length and size of the output of each task shown in Fig. 3.1(a) are measured in Million Instructions (MI) and GigaBytes (GB) respectively. The tasks are to be mapped to two Grid resources R_1 and R_2 with processing capability (PC) and transfer capacity, i.e. Bandwidth (BW) as indicated at the bottom of Fig. 3.1.

First, the AET and $ADTT$ values for each task are calculated as shown in Fig. 3.1(a). Then using these values, $AEST$ and $ALST$ of all the tasks are calculated according to Section 3.3.1 (see Fig. 3.1(b)). Since T_0, T_2, T_3, and T_4 have equal $AEST$ and $ALST$, they are on critical path with T_0 as the highest task. Hence, T_0 is selected as the critical task and mapped to resource R_1, which gives T_0 the minimum combined start time. At the end of this step, the schedule length of the workflow (i.e. $DCPL$) is 890. Similarly, in Fig. 3.1(c), T_2 is selected as the critical task and mapped to R_1. As both T_0 and T_1 are mapped to R_1 and the data transfer time of T_0 is now zero, the $AEST$ and $ALST$ of all the tasks are changed and the schedule length becomes 850 (Fig. 3.1(d)). In the next step, T_3 is mapped to R_1 as well and the $DCPL$ is reduced to 770 as the data transfer time for T_2 is zero.

Now T_4 is the only task remaining on the critical path (Fig. 3.1(e)). However, one of its parent tasks, T_1, is not mapped yet and therefore T_1 is selected as the critical task. As T_2 and T_3 are already mapped to R_1, the start time of T_1 on R_1 is 700. Therefore, T_1 is mapped to R_2 as its start and end times on R_2 are 180 and 430 respectively. Finally, when T_4 is mapped to R_1 (Fig. 3.1(g)), all the tasks have been mapped, the schedule length can not be improved any further, and a schedule length of 750 is obtained. The final schedule generated by DCP-G is shown in a table in Fig. 3.1(h).

3.4 Performance Evaluation

We evaluate DCP-G by comparing the schedules produced by it against those produced by the other algorithms described previously for a variety of workflows in a simulated Grid

environment. In this section, first we describe our simulation methodology and setup, and then present the results of experiments.

3.4.1 Simulation methodology

We use GridSim [22] toolkit to simulate the application and Grid environment for our simulation. The GridSim toolkit allows modeling and simulation of various entities in parallel and distributed computing environment, such as systems-users, applications, resources, and resource brokers (schedulers) for design and evaluation of scheduling algorithms. It provides a comprehensive facility for creating different types of heterogeneous resources for executing compute and data intensive applications. A resource can be a single processor or multi-processor with shared or distributed memory and managed by time or space shared schedulers. The processing nodes within a resource can be homogeneous or heterogeneous in terms of processing capability, configuration, and availability. We model different entities of GridSim in the following manner.

Workflow model

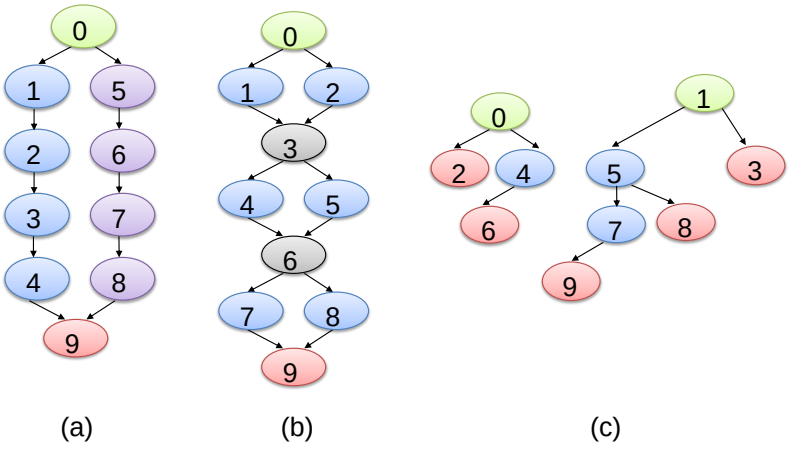


Figure 3.2: Three sample workflows: (a) Parallel workflow; (b) Fork-join workflow; (c) Random workflow.

We implement a workflow generator that can generate various formats of weighted pseudo-application workflows. The following input parameters are used to create a workflow.

- N, the total number of tasks in the workflow.
- α, the shape parameter represents the ratio of the total number of tasks to the width (i.e. maximum number of nodes in a level). Thus, width $W = \lceil \frac{N}{\alpha} \rceil$.
- Type of workflow: Our workflow generator can generate three types of workflow namely parallel workflow, fork-join workflow, and random workflow.

Parallel workflow: In parallel workflows [120], a group of tasks creates a chain of tasks with one entry and one exit task; there can be several such chains in one workflow. Here, one task is dependent on only one task, although the tasks at the head of chains are dependant on the entry task, and the exit task is dependant on the tasks at the tail of chains. Number of levels in a parallel workflow can be specified as:

Number of levels = $\lfloor \frac{N-2}{W} \rfloor$

Fork-join workflow: In fork-join workflows [18], forks of tasks are created and then joined. So, in this kind of workflows, there can be only one entry task and one exit task, but the number of tasks in each level depends on the total number of tasks and width of that level, W. Number of levels in a fork-join workflow can be specified as,

Number of levels = $\lfloor \frac{N}{W+1} \rfloor$

Random workflow: In random workflows, the dependency and number of parent tasks of a task, which equals to the indegree of a node in DAG representation of the workflow, is generated randomly. Here, the task dependency and the indegree are calculated as,

Maximum Indegree $(T_i) = \lfloor \frac{W}{2} \rfloor$
Minimum Indegree $(T_i) = 1$
Parent $(T_i) = \{T_x | T_x \epsilon [T_0 .... T_{i-1}]\}$; if T_i is not a root task.
where x is a random number and $0 <= x <= \lfloor \frac{W}{2} \rfloor$.
Parent $(T_i) = \{\phi\}$; if T_i is a root task.

Table 3.3: Resources used for performance evaluation

Resource name/site	Location	Number of nodes	Single PE rating (MIPS)	Mean load
RAL	UK	41	1140	0.9
NorduGrid	Norway	17	1176	0.9
NIKHEF	Netherlands	18	1166	0.9
Milano	Italy	7	1000	0.5
Torino	Italy	4	1330	0.5
Catania	Italy	5	1200	0.6
Padova	Italy	13	1000	0.4
Bologna	Italy	20	1140	0.8

In Fig. 3.2, a sample of each type of workflow is illustrated, where $N = 10$ and $\alpha = 5$. In simulation, we use MI (Million Instruction) to denote the length of tasks and MB (Mega Byte) to denote the output data size of each task.

Resource model

As the execution environment for tasks in scientific workflows is heterogeneous, we use heterogeneous resources with different processing capabilities. Here, we choose 8 resources (refer to Table 3.3) from the European Data Grid (EDG) 1 test bed [16] used for simulation in [125]. The processing capability of the resources is measured in MIPS (Million Instructions per Second) and the bandwidth in Mbps (Megabits per second).

3.4.2 Simulation setup

The workflows for evaluation are generated using the following parameters:

- Type = parallel, fork-join, random
- $N = \{50, 100, 200, 300\}$
- $\alpha = \{10\}$

Here, size of each task in the workflow is generated from a uniform distribution between $100,000$ MI to 500000 MI, while the output data size of each task is also generated from a uniform distribution between 1 GB and 5 GB.

For GRASP, we run 600 iterations to map tasks to resources, and then we select the best schedule out of the generated schedules. For GA, parameters for various genetic operators, such as selection, crossover, and mutation are set using those applied in previous

Table 3.4: Parameters of Genetic Algorithm

Parameter	Value/Type
Population size	60
Crossover probability	0.7
Swapping mutation probability	0.5
Replacing mutation probability	0.8
Fitness function	Makespan of workflow
Selection scheme	Elitism-Roulette Wheel
Stopping condition	300 iterations
Initial individuals	Randomly generated

studies [135]. Table 3.4 shows the values of different parameters used for simulating GA.

3.4.3 Results and Observations

We evaluate the scheduling heuristics based on the total makespan produced and the time required for scheduling workflows. Makespan or workflow completion time is the total time required for executing all the tasks of a workflow, where the completion time of a task is the difference between the time when a task has been submitted by the scheduler to a resource and the time when the output of that task has been achieved.

Two sets of experiments were carried out. In the first set, we consider an ideal case, where the availability and load of Grid resources remain static over time. For this environment, we statically map tasks to resources according to different strategies and execute tasks accordingly. In the next set, we evaluate the strategies in a more realistic scenario, where the availability and load of Grid resources vary over time. In this case, the instantaneous load (i.e. number of PEs occupied) for each resource during the simulation is derived from a Gaussian distribution, as performed in [125].

Execution time (Makespan) in static environment

The graph in Fig. 3.3 plots the execution time of parallel, fork-join, and random workflows of 50, 100, 200, and 300 tasks for seven workflow scheduling strategies namely, Myopic, Min-Min, Max-Min, HEFT, DCP-G, GRASP, and GA in static environment.

For random workflow (refer to Fig. 3.3(c)), DCP-G can generate schedules with up to 13% less makespan than HEFT, which generates better schedule than Myopic, Min-min,

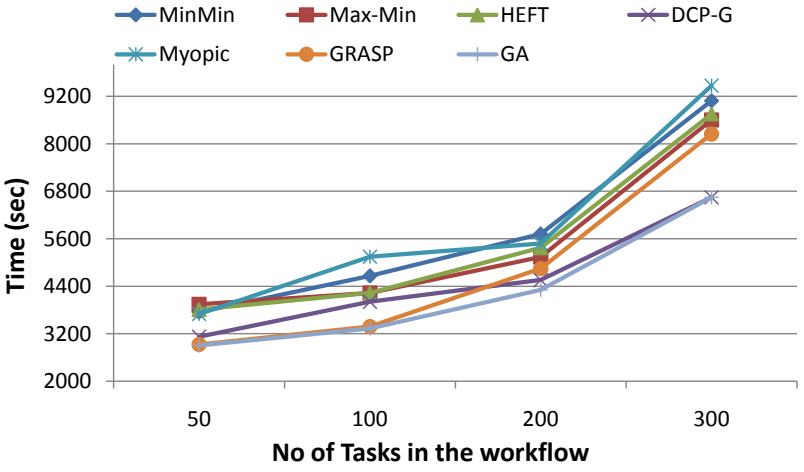
and Max-min. Since from any task in the random workflow, there can be multiple paths to an exit node, dynamically assigning priorities to tasks helps DCP-G to generate better schedules. As GRASP and GA search the entire solution space for the best schedule, they generate $20 - 30\%$ better schedule than DCP-G.

However, the execution time of fork-join workflows (refer to Fig. 3.3(b)) shows a significant difference between heuristic and meta-heuristic based approaches. During the process of task selection for mapping, heuristic-based approaches do not consider the impact of mapping child tasks. Thus, all the heuristic based techniques generate similar schedule with DCP-G being marginally better. However, in a fork-join workflow, a join task depends on the output of all the forked independent tasks that precede it. If this join task is assigned to a resource with low bandwidths to other resources, increase in data transfer time impacts the makespan adversely. However, meta-heuristics (GA, GRASP) consider the impact of mapping not only the parent fork tasks but also the child join tasks, and are therefore able to generate $40 - 50\%$ better schedule than DCP, which is the best among heuristic-based methods.

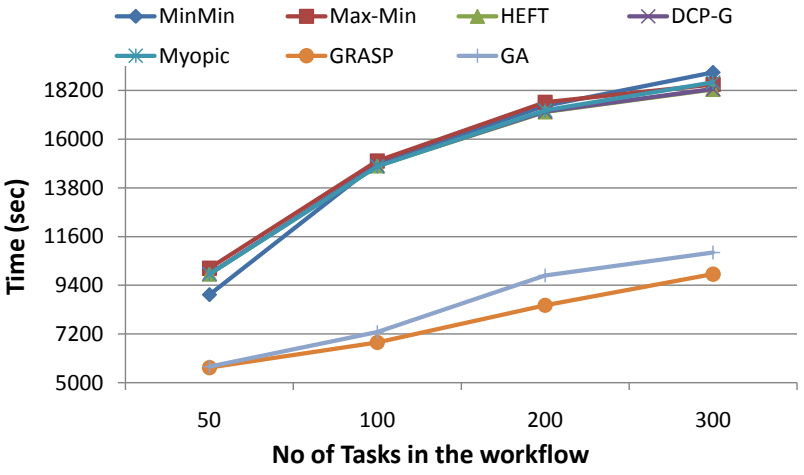
According to Fig. 3.3(a), the execution time of parallel workflow exhibits a slow exponential growth with the increase in workflow size. The reason is that, unlike fork-join workflows, the number of unmapped ready tasks at every step of scheduling in a parallel workflow is always equal to W, and a task becomes ready as soon as its parent finishes. Thus, when the available resources are less than unmapped ready tasks, the time spent by some of these tasks in waiting to be scheduled results in an increase in the total execution time. In case of parallel workflows, DCP-G and GA generate better schedules than others, and the makespan is reduced by at least 20%. Here, the execution time of GRASP rises beyond that of DCP-G as the number of candidate solutions for task mapping increases exponentially with the workflow size.

Execution time (Makespan) in dynamic environment

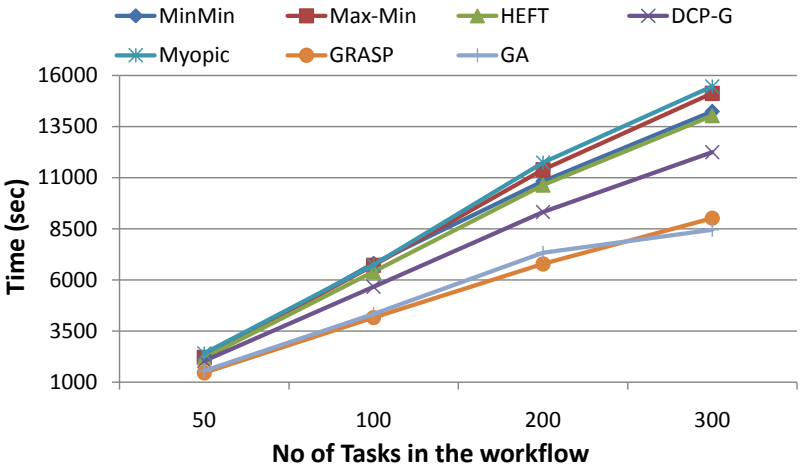
As the resource availability changes over time in dynamic environment, the resource availability information needs to be continuously updated after a certain period of time, and the tasks have to be remapped if necessary, depending on the updated availability of resources. Here, we compare the performance of rescheduling using DCP-G and other



(a) Parallel workflow

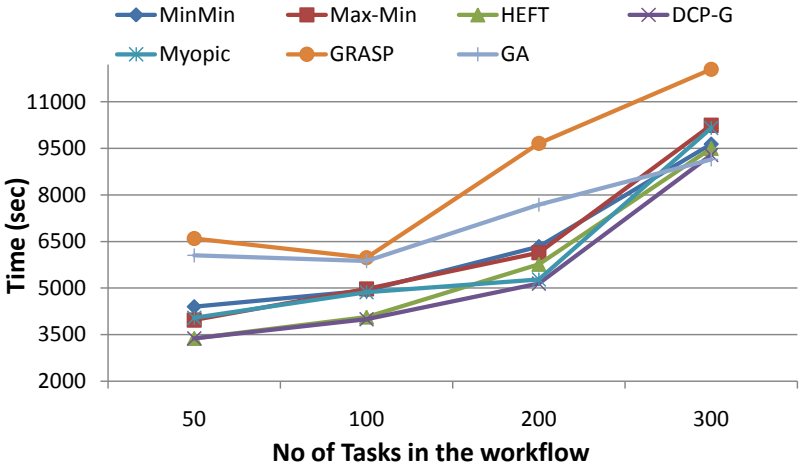


(b) Fork-join workflow

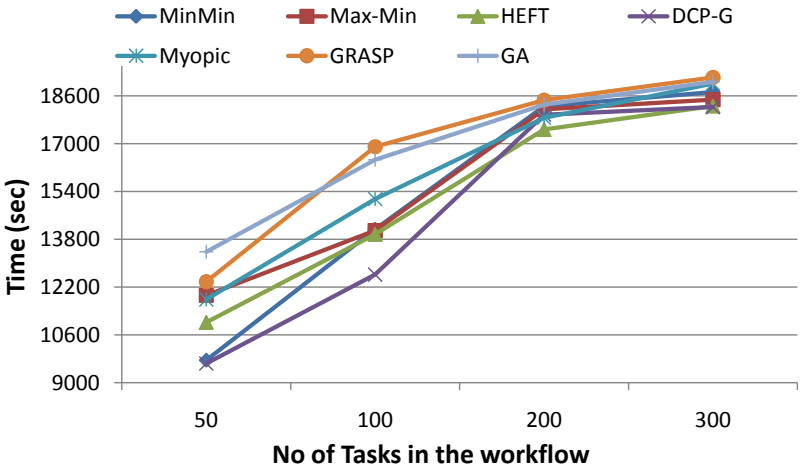


(c) Random workflow

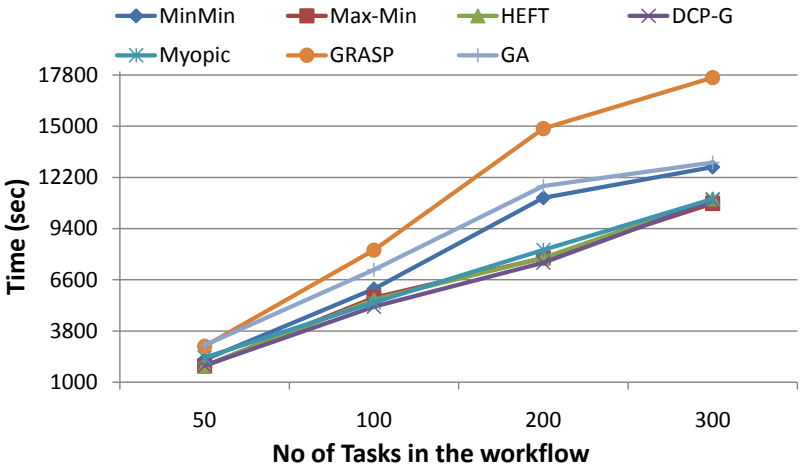
Figure 3.3: Execution time of different type of workflows for static environment.



(a) Parallel workflow



(b) Fork-join workflow



(c) Random workflow

Figure 3.4: Execution time of different type of workflows for dynamic environment.

heuristic-based approaches against the static schedules generated by the meta-heuristics.

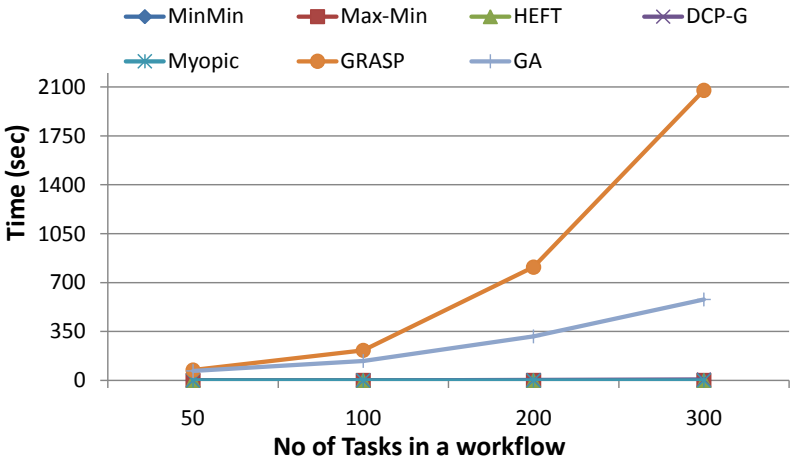
Fig. 3.4 shows the execution time of different scheduling techniques in dynamic environment, where resource information is updated every 50 seconds. The number of available processing elements, and hence the number of tasks that can start execution in a resource varies with the load on the resource. However, for GA and GRASP, if a resource is heavily-loaded and unavailable, the tasks mapped to that resource have to wait to be executed. This waiting time consequently impacts the start time of other dependent tasks and increases the makespan. This is reflected in the poor performance of GA and GRASP in the graphs in Fig. 3.4, where heuristic-based approaches generate up to 30% better schedules than these two meta-heuristic based approaches. Among the heuristics, DCP-G is able to achieve up to 6% better makespan than the others. This is because, in DCP-G, tasks on the critical path waiting to be executed on a heavily-loaded resource are rescheduled to resources with available PEs. This reduces the critical path length; thus, the makespan for workflow execution is also reduced.

It can also be seen that heuristic-based approaches perform better in dynamic than static environments for the same workflows and experimental setup. This can be attributed to the fact that not only the load but the resource availability in dynamic environments is updated regularly. This means that the heuristics are able to adapt to resources that are more frequently available and therefore, produce better schedules.

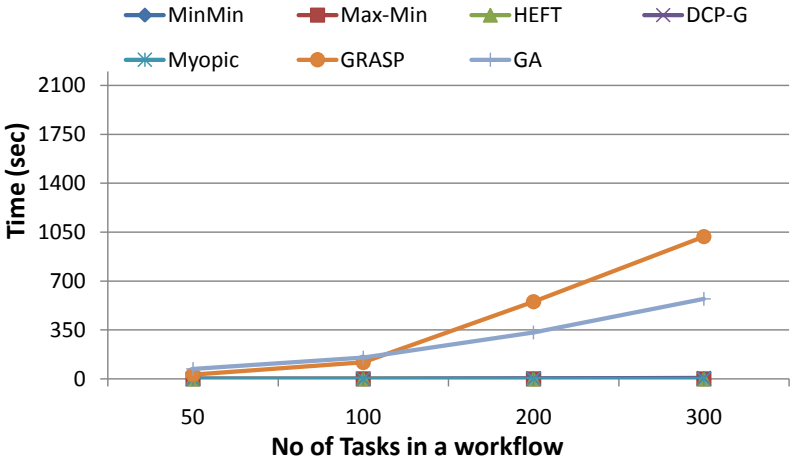
Scheduling time

Fig. 3.5 shows the total scheduling time of a workflow for different scheduling techniques in case of three types of workflow. Scheduling time is considered as scheduling overhead, which constitutes a significant amount of time used by scheduler to generate schedule.

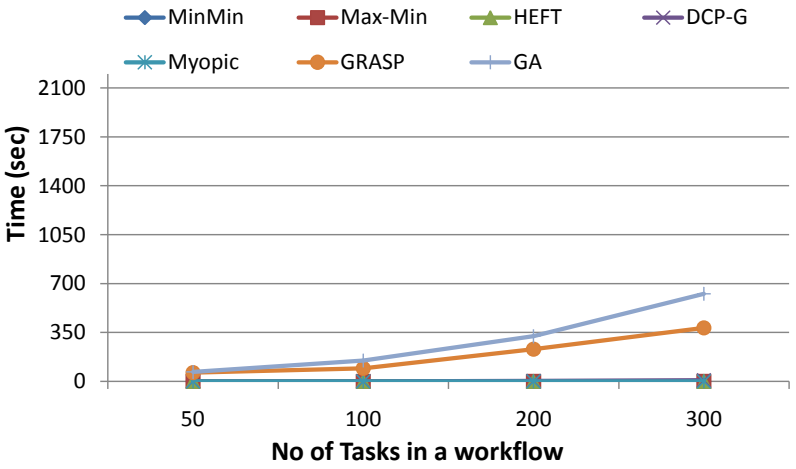
For the convenience of discussion, the average scheduling time (in milliseconds) for one task of parallel, fork-join, and random workflows to generate a single schedule for different scheduling techniques is presented in Table 3.5. To generate a single schedule, Myopic, Min-Min, Max-Min, and HEFT require nearly 1 millisecond for each task irrespective of the workflow size and type, whereas the average scheduling time of one task for DCP-G is 16 to 17 milliseconds, and does not vary with the workflow type as the task selection procedure is independent of workflow structure.



(a) Parallel workflow



(b) Fork-join workflow



(c) Random workflow

Figure 3.5: Scheduling time of different scheduling approaches for various type of workflows.

Table 3.5: Average scheduling time per task

Scheduling strategy	Random workflow(ms)	Fork-join workflow(ms)	Parallel workflow(ms)
Myopic	1	1	1
Min-Min	1	1	1
Max-Min	1	1	1
HEFT	1	1	1
DCP-G	17	16	16
GRASP	1180	2840	5720
GA	1940	1780	1750

Scheduling time using GRASP increases exponentially not only with the increase of tasks in a workflow but also with change in workflow structure. In each iteration, GRASP creates Restricted Candidate List (RCL) for each unmapped ready task and then selects a resource for the task randomly. When number of tasks increases, RCL increases exponentially resulting in increased scheduling time. But the size of RCL is also dependent on workflow structure. For example, when a workflow consists of 300 tasks, parallel and fork-join structures contain 30 tasks in each level, whereas the random structure contains random number of levels as well as random number of tasks in each level. Thus, at every step a parallel workflow has 30 ready tasks, fork-join workflow has maximum 30 ready tasks, and average number of ready tasks in each level of random workflow is less than 30. Therefore, the scheduling time for random workflow is the lowest, and parallel workflow is the highest in this case.

However, scheduling time for GA does not change much with the workflow type because it executes the same number of genetic operations irrespective of workflow structure. But the size of each individual in the solution space is equal to number of tasks in workflow. Thus, scheduling time increases with the increase in the size of workflow.

While it is possible to reschedule at regular intervals in GA and GRASP, Table 3.5 shows that the scheduling times for these are at least 100 times as high as DCP-G and increase with the size of workflow as well. Hence, we did not incorporate rescheduling for GA and GRASP in the experiments for the dynamic environment.

3.4.4 Discussion

From Fig. 3.3, it is evident that among the heuristic-based scheduling techniques, DCP-G can generate better schedule by up to 20% in static environment, especially for random

and parallel workflows, irrespective of workflow size. GA and GRASP can generate more effective schedule than DCP-G for random and fork-join workflow, but they suffer from the problem of higher scheduling time. In our simulation, for parallel workflow of 300 tasks, DCP-G takes 6 seconds to map the tasks to resources, whereas GA and GRASP take 580 and 2076 seconds respectively.

In dynamic environment, heuristics based techniques adapt to the dynamic nature of resources and can avoid performance degradation. But meta-heuristic based techniques perform worse in this situation due to the unavailability of mapped resources at certain intervals. However, in dynamic environment, DCP-G can generate better schedule than other approaches, irrespective of workflow type and size.

3.5 Conclusions

In this chapter, we have defined the workflow scheduling problem in Grid computing environment and discussed the existing well known workflow scheduling techniques. Nevertheless, these techniques are static in nature and do not take into account dynamic resource behavior. Therefore, we have proposed a dynamic scheduling approach, named DCP-G for scheduling Grid workflows. DCP-G determines an efficient mapping of workflow tasks to Grid resources by calculating the critical path in the workflow task graph at every step and assigns priority to a task in the critical path that is estimated to complete earlier. We have compared the performance of DCP-G with other existing heuristic and meta-heuristic based scheduling strategies for different types and sizes of workflows. The results show that DCP-G can generate better schedule for most of the workflow types, irrespective their sizes particularly when the resource availability changes frequently.

In summary, this chapter identifies that dynamic scheduling approaches can adapt to temporal behaviour of heterogeneous Grid resources and are able to avoid performance degradation by generating efficient schedules. Thus, in the next chapters, we use dynamic scheduling heuristics to achieve self-adapting environment for facilitating autonomic workflow management.

Chapter 4

Autonomic Workflow Management System Architecture

In this chapter, we present the architectural framework for autonomic workflow management system in Grids. First, we describe the system models, such as Grid model, coordination model, and application model that form the basis of this architectural framework. We utilize the Grid Federation model for developing our Grid model, which essentially specifies the policies and protocols for resource organization and Grid networking. Then, we present the coordination model that outlines the communication, coordination, and indexing protocols to form the basis for coordinated application scheduling among the distributed Grid sites. Next, The application model is provided that specifies how the workflow application is modeled and integrated into the proposed workflow management system.

Furthermore, the prototype implementation of the proposed architecture is also discussed in this chapter. The prototype wokflow management system is developed by leveraging Aneka enterprise Grid platform that enables the creation of Aneka Federation, a decentralized Grid resource sharing model. Aneka Federation realizes the theoretical Grid resource sharing model, Grid Federation. The main component of Aneka Federation that enables the transparent resource sharing among Aneka enterprise Grids is called Aneka Coordinator. The Grid Autonomic Manager (GAM), discussed in Section 4.1.1 is implemented in the prototype system by Aneka Coordinator, which manages the resource

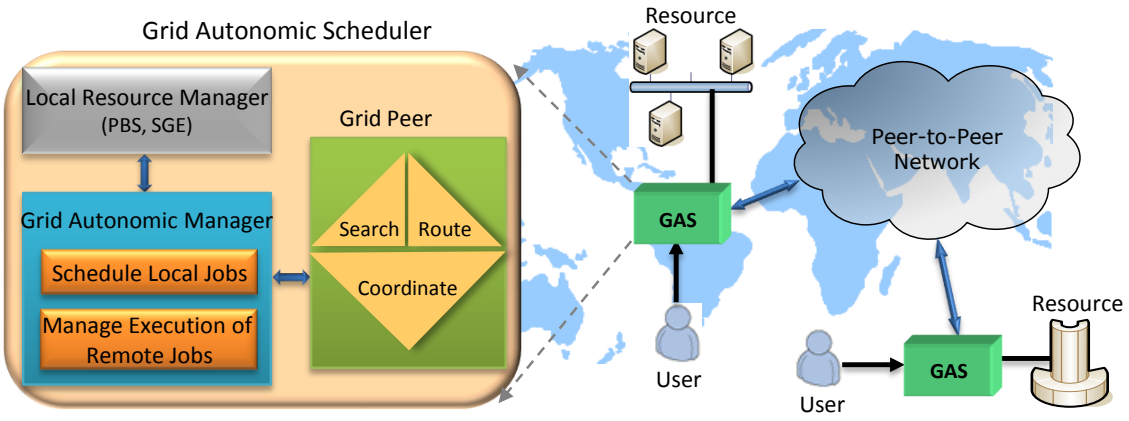


Figure 4.1: Grid Federation.

discovery and cooperative task scheduling using a DHT-based decentralized P2P overlay. In addition, the required components and services of the prototype system, such as application development, resource discovery, coordination, and task scheduling are also illustrated with the direction for deployment of the system.

4.1 Architectural Framework

4.1.1 Grid Model

The proposed architecture for autonomic workflow management system in Grids utilizes the *Grid Federation* (GF) [101] model in regards to resource organization and Grid networking. Grid Federation aggregates distributed resource brokering and allocation services as part of a cooperative resource sharing environment. Table 4.1 shows the notations for resource, workflow, and coordination models.

Definition 4.1 (Grid Federation): The Grid Federation, $G_F = \{S_1, S_2, ..., S_n\}$ *consists of a number of sites* n*, with each site contributing its resource to the Grid. Every site in Grid Federation has its own resource description* D_i*, which contains definition of the resource that it is willing to contribute.* D_i *includes information about the CPU architecture, number of processors, memory size, secondary storage size, operating system type, etc.*

In this work, $D_i = (p_i, a_i, s_i, o_i)$, which includes the number of processors p_i, proces-

Table 4.1: Notations: resource, workflow, and coordination models

Symbol	Meaning
Resource	
r	number of resources or GASs in the Grid system.
D_i	configuration of the i-th resource in the system.
R_i	i-th Resource in the system.
G_i	i-th GAS in the system.
S_i	processor speed at GAS i.
P_i	number of processors for resource at GAS i.
A_i	processor architecture for resource at GAS i.
O_i	operating system type for resource at GAS i.
Workflow	
$W_{i,j,k}$	i-th workflow from the j-th user of k-th GAS in the system.
$T_{x_{i,j,k}}$	x-th task of the workflow, $W_{i,j,k}$.
$s_{x_{i,j,k}}$	processor speed required by the task $T_{x_{i,j,k}}$.
$p_{x_{i,j,k}}$	number of processors required by the task $T_{x_{i,j,k}}$.
$a_{x_{i,j,k}}$	processor architecture required by the task $T_{x_{i,j,k}}$.
$o_{x_{i,j,k}}$	operating system required by the task $T_{x_{i,j,k}}$.
Coordination	
$r_{x_{i,j,k}}$	a claim posted for task $T_{x_{i,j,k}}$.
u_i	a ticket issued by the i-th GAS/broker.
dim	dimensionality or number of attributes in the Cartesian space.
f_{min}	minimum division level of d-dimensional index tree.

sor architecture a_i, their speed s_i, and installed operating system type o_i.

The application scheduling and resource discovery in GF are facilitated by a specialized Grid Resource Management System (GRMS), known as Grid Autonomic Scheduler (GAS). Fig. 4.1 shows an example GF resource sharing model consisting of Internet-wide distributed Grid sites. Every contributing Grid site in GF maintains its own GAS service. A GAS service is composed of three software components: Grid Autonomic Manager (GAM), Local Resource Management System (LRMS), and Grid Peer.

The GAM component of GAS exports a Grid site to the outside world and is responsible for scheduling locally submitted jobs (workflows, parallel applications) in the Grid. Further, it also manages the execution of remote jobs (workflows) in conjunction with LRMS. The LRMS software module can be realized by leveraging systems, such as SGE (Sun Grid Engine) [51] and PBS [20]. Additionally, LRMS performs other activities for facilitating Grid wide job submission and migration process, such as answering the GAM queries related to local job queue length, expected response time, and current resource utilization status.

Grid Federation requires supporting technologies to enable scalable collaboration and

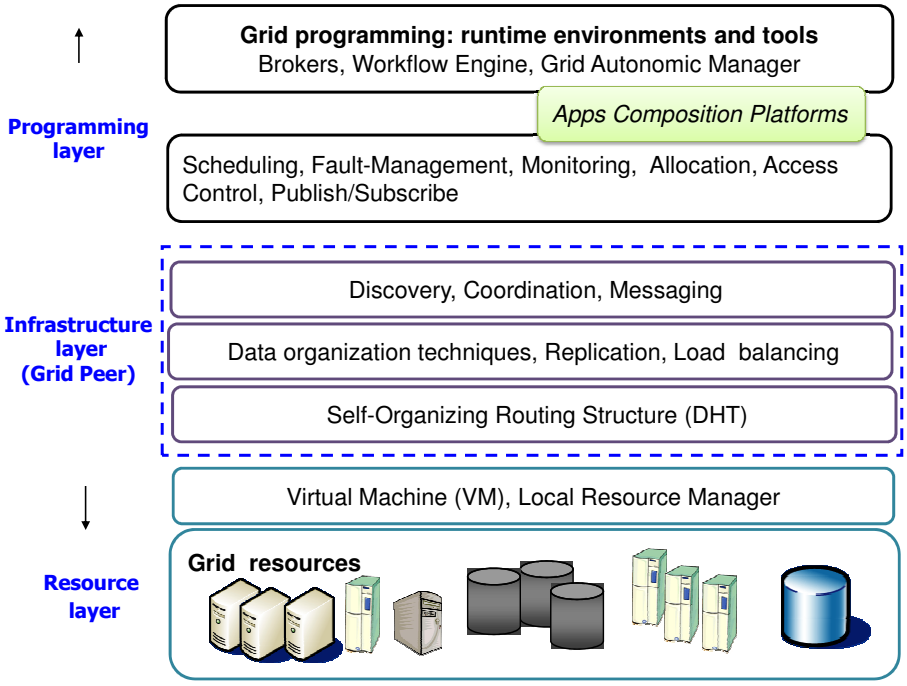


Figure 4.2: Layerd design of autonomic workflow management system architecture

communication between resources and services across multiple Grid sites. For supporting the aforementioned functions, it is mandatory to build some kind of overlay network (as implemented in CometCloud [67]) on top of the physical routing network. To this end, the Grid peer (see Fig. 4.2) implements an overlay (infrastructure level core services) for enabling decentralized and distributed resource discovery that supports efficient resource status lookups and updates across the federation. It also enables decentralized inter-GAM collaboration for optimizing load-balancing and distributed resource provisioning. Similar to CometCloud, these core services are divided into a number of sub-layers (refer to Fig. 4.2): (i) higher level services for discovery, coordination, and messaging; (ii) low level distributed indexing and data organization techniques; and (iii) self-organizing overlay that builds over Distributed Hash Table (DHT) [115] [105] routing structure.

4.1.2 Coordination Model

In this section, we first describe the communication, coordination and indexing models that are utilized to facilitate the P2P coordination space. Then, we present the composition of objects, access primitives that form the basis for coordinating application schedules

among distributed GASs/brokers.

Layered Design of the Coordination service

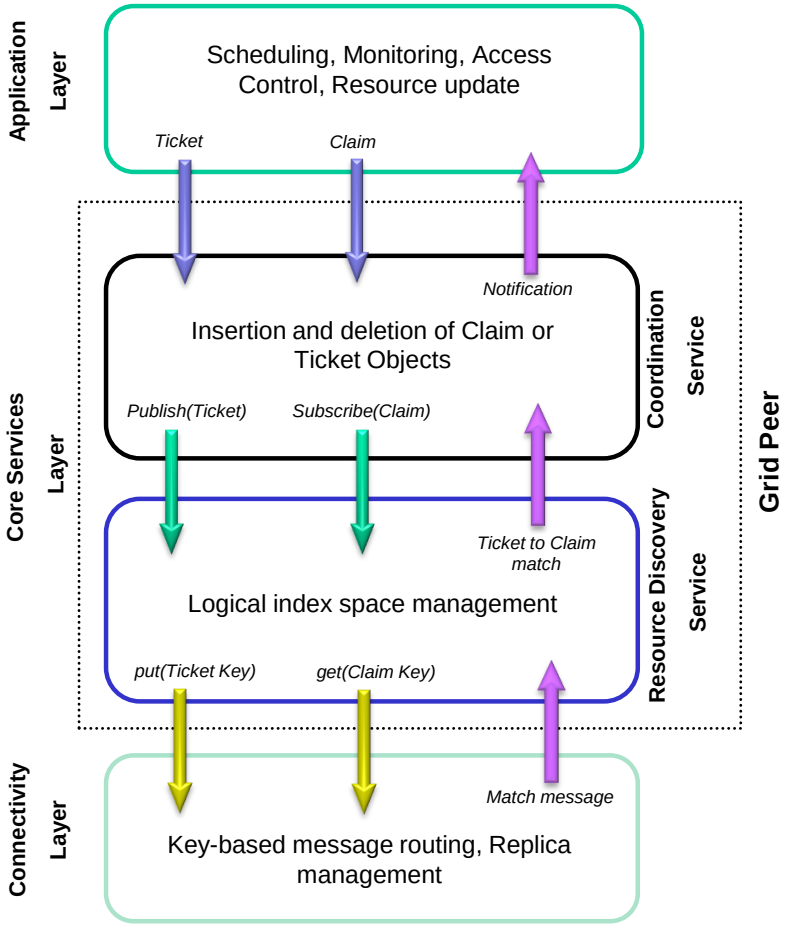


Figure 4.3: Layered design of the core services.

We design and architect the coordination service based on a layered approach. The architecture consists of three layers: the *Application* layer, *Core Services* layer and *Connectivity* layer. Grid Services, such as workflow brokers/schedulers work at the Application layer and insert objects (claims/tickets) via the Core Services layer. In the context of a GAS, the GAM software module operates at the Application layer.

We use the coordination service as a sub-layer of the Core Services layer. The coordination service accepts the application objects, such as claims and tickets. These objects are similar to Condor classified advertisements (ClassAds) [98] and encapsulate the co-

ordination logic, which in this case is the resource provisioning logic. These objects are managed by the coordination service. The calls between the coordination service and resource discovery service are made through the standard publish/subscribe technique. The resource discovery service is responsible for managing the logical index space and communicating with the Connectivity layer. The calls between Core Services layer and Connectivity layer are made through standard DHT primitives, such as *put()* and *get()* that are defined in the P2P Common Application Programming Interface specification [35]. As shown in Fig. 4.3, Core Services layer is managed by the Grid peer module of a GAS service.

The Connectivity layer is responsible for undertaking key-Based routing in the DHT space, such as Chord, CAN, Pastry etc. In this chapter, for simulation, we model the Chord substrate at the Connectivity layer. Chord hashes the peers and objects (such as fileIds, logical indices, etc) to the circular identifier space and the average lookup complexity is $O(\log r)$ steps with high probability. Each peer in the Chord network is required to maintain routing table state of $O(\log r)$ other peers, where r is the total number of Grid peers in the system.

Coordination Objects

This section gives details about the resource claim and ticket objects that form the basis for enabling decentralized coordination mechanism among the GASs in Grid Federation system. These coordination objects include Resource Claim and Resource Ticket.

Every GAS in the federation posts its resource ticket through the local coordination service. A resource ticket object u_i or update query consists of a resource description R_i, for a resource i.

Example 4.2 (Resource Ticket): Total-Processors = 100 && Processor-Arch= Pentium && Processor-Speed= 2 GHz && Operating-System = Linux && Utilization=0.80.

A resource claim or lookup object encapsulates the resource configuration requirements of a task in the workflow submitted by users. In this work, we focus on the workflows for which the requirements are confined to a computational Grid or PlanetLab resources. Users submit their workflow application's resource requirements to the local GAS. The corresponding Grid peer service of GAS is responsible for searching the

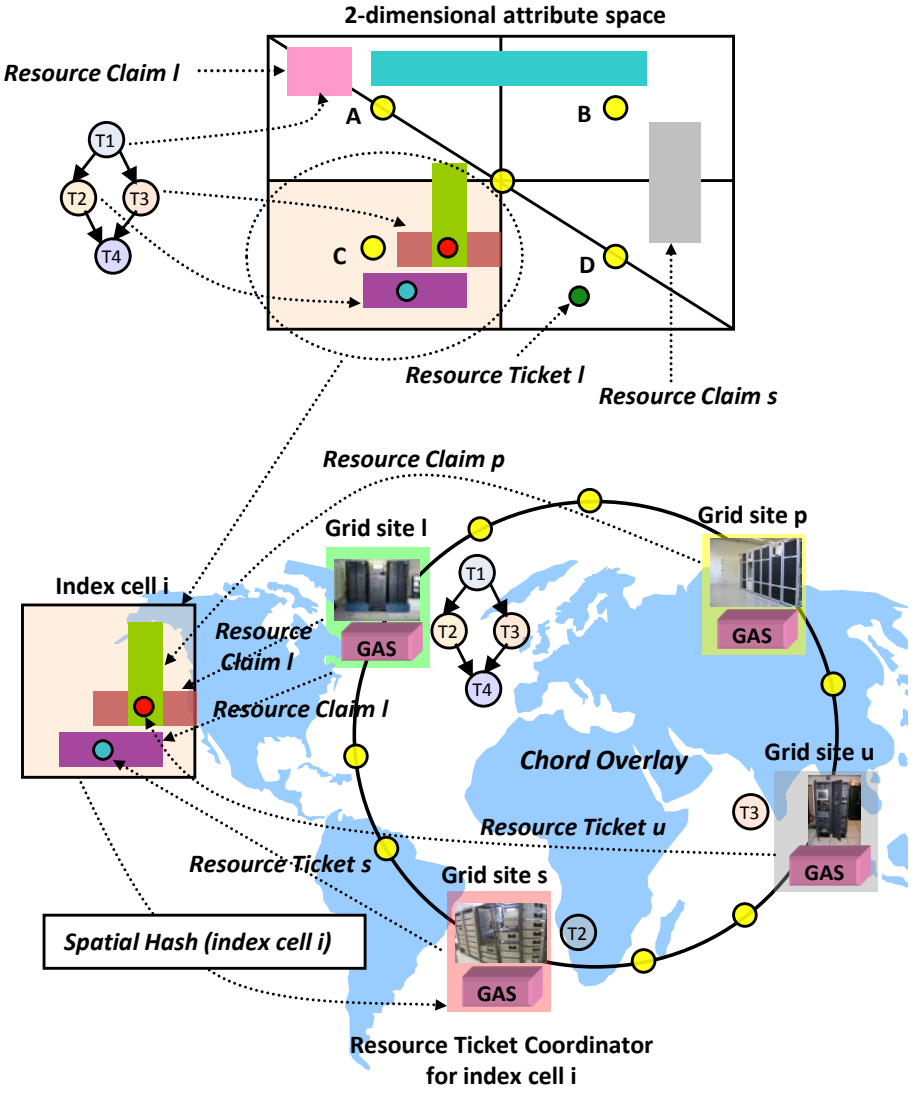


Figure 4.4: Resource allocation and application scheduling coordination across Grid sites.

suitable resources in the federated system. The GAS aggregates characteristics of a task including the number of processors, processor architecture, and installed operating system with constraint on maximum speed and resource utilization into a resource claim object, $r_{i,j,k}$.

Example 4.3 (Resource Claim): Total-Processors $\geq$ 70 && Processor-Arch= Pentium && 2 GHz $\leq$ Processor-Speed $\leq$ 5GHz && Operating-System = Solaris && 0.0 $\leq$ Utilization $\leq$ 0.90.

The GAM module of a GAS passes the resource ticket and claim object to the coordination service operating at the Core Services layer. Recall that, the Core Services layer is managed by the Grid peer module, which interacts with the DHT based overlay network. The operation between Grid peer module and DHT based overlay is transparent to GAM. In other words, a GAM is not aware of how the Grid peer module is routing, searching, and matching the objects in the system. It is the responsibility of a Grid peer module to implement specific communication and data organization methods, which can provide desired functionality to the Application layer services, such as a GAM. In order to efficiently route, search, and match the d-dimensional claim and ticket objects, the Grid peer module embeds a logical spatial data-structure over the DHT overlay space.

The resource ticket and claim objects are spatially hashed to an index cell i in the d-dimensional coordination space. Similarly, the coordination services of the resource sites in the Grid network hash themselves into this coordination space using the overlay hashing function (SHA-1 in case of Chord and Pastry). In Fig. 4.4, resource claim objects issued by site p and l are mapped to the index cell i and these claim objects are currently hashed to the site s. Thus, site s is responsible for coordinating the resource sharing among all the resource claims that are mapped to the cell i. Subsequently, site u issues a resource ticket (shown as small dark circles in Fig. 4.4) that falls under a region of the space currently required by users at site p and l. In this case, the coordination service of site s has to decide, which of the sites (i.e. either l or p or both) will be allowed to claim the ticket issued by site u. This load-distribution decision is based on the fact that it should not lead to over-provisioning of resources at site u.

In Table 4.2, we show an example list of claim objects generated by scheduler for the corresponding workflow tasks listed in Table 4.3 that are stored with a coordination

Table 4.2: Claims stored with the coordination service at time τ

Time	Claim ID	$S_{x_{i,j,k}}$	$P_{x_{i,j,k}}$	$A_{x_{i,j,k}}$	$O_{x_{i,j,k}}$	Rank
200	Claim 1	> 800	1	Intel	Linux	0.2
350	Claim 2	> 1200	1	Intel	Linux	0.3
500	Claim 3	> 700	1	Sparc	Solaris	0.1
700	Claim 4	> 1500	1	Intel	Windows XP	0.4

Table 4.3: Resource configuration required for workflow tasks

Task No	Claim ID	$S_{x_{i,j,k}}$	$P_{x_{i,j,k}}$	$A_{x_{i,j,k}}$	$O_{x_{i,j,k}}$
1	Claim 1	> 800	1	Intel	Linux
2	Claim 2	> 1200	1	Intel	Linux
3	Claim 3	> 700	1	Sparc	Solaris
4	Claim 4	> 1500	1	Intel	Windows XP

Table 4.4: Ticket published to the coordiation service at time τ

Time	GAS ID	μ_i	p_i	p_i avail	x_i	ϕ_i
900	GAS-8	1400	3	2	Intel	Linux

service at time $t = 900$ seconds. Essentially, the claims in the list arrived at a time $\leq$ 900 and are waiting for a suitable ticket object that can meet its resource configuration requirements. Whereas, Table 4.4 depicts the list of ticket objects that have arrived at $t = 900$ seconds. Following the ticket arrival event, the coordination service undertakes a procedure that divides this ticket object among the list of claims. Based on the resource attribute specification and availability, only Claim 1 and Claim 2 matches the ticket's resource configuration. As specified in the ticket object, there are currently 2 processors are available with the GFA 8, which is equal to the sum of processors required by Claim 1 and 2 (i.e., 2). Hence in this case the coordination service, based on the priority of the task (rank), first notifies the GFA that has posted Claim 2 and follows it with the GFA responsible for Claim 1. However, Claims 3 and 4 have to wait for the arrival of tickets that can match their required resource configuration.

Once a resource ticket matches with one or more resource claims, a coordination service sends *notification* messages to the resource claimers such that it does not lead to

the overloading of the concerned resource ticket issuer. Thus, this mechanism prevents the workflow brokers from overloading the same resource. As a result, the problem of conflicting schedules is overcome.

D-dimensional Coordination Object Mapping and Routing

one-dimensional hashing, provided by current implementation of DHTs are insufficient to manage complex objects, such as resource tickets and claims. DHTs generally hash a given unique value/identifier (e.g. a file name) to a 1-dimensional DHT key space and hence they cannot support mapping and lookups for complex objects. Management of these objects whose extents lie in the d-dimensional space requires embedding of a logical index structure over the 1-dimensional DHT key space [103].

We now describe the features of the P2P-based spatial index that we utilize for mapping the d-dimensional claim and ticket objects over the DHT space. Providing the background and details on this topic is beyond the scope of this thesis; here we only give a high level view. The spatial index that we consider in this work, assigns regions of space to the Grid peers in the Grid Federation system. If a Grid peer is assigned a region of d-dimensional space, then it is responsible for handling query computation associated with the claim and ticket objects that intersect this region, as well as storing the objects that are associated with the region. Fig. 4.5 presents a 2-dimensional Grid resource attribute space for mapping claim and ticket objects. The attribute space has a grid-like structure due to its recursive division process. The index cells, resulted from this process, remain constant throughout the life of the d-dimensional attribute space and serve as the entry points for subsequent mapping of claim and ticket objects. The number of index cells produced at the minimum division level, f_{min} is always equal to $(f_{min})^{dim}$, where dim is the dimensionality of the Cartesian space. These index cells are called base index cells and they are computed when the Grid peers bootstrap to the coordination network. Finer details on recursive sub-division technique can be found in [116]. Every Grid peer in the network has the basic information about the Cartesian space coordinate values, dimensions and minimum division level.

Every cell at the f_{min} level is uniquely identified by its centroid, termed as a *control point*. Fig. 4.5 depicts four control points A, B, C and D. A DHT hashing method such

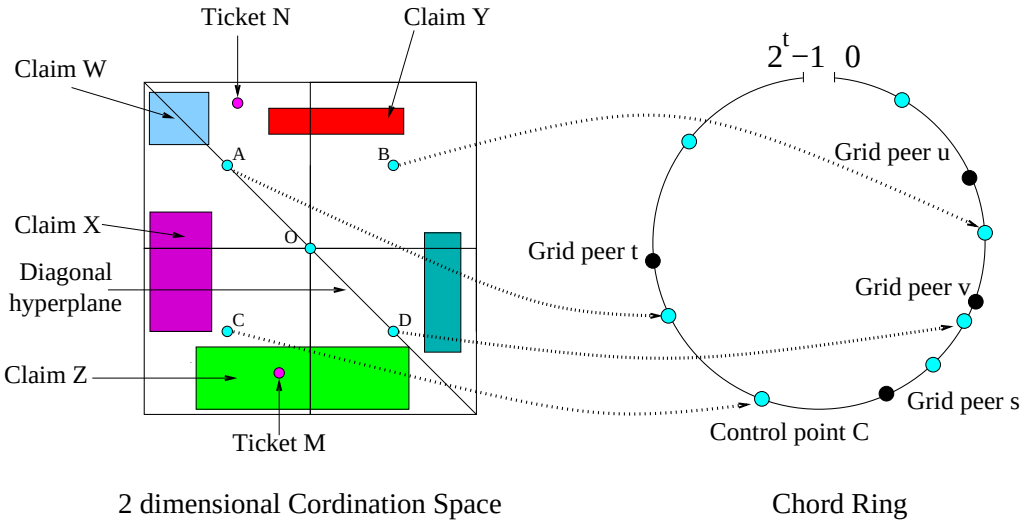


Figure 4.5: Spatial resource claims $\{W,X,Y,Z\}$, cell control points $\{A,B,C,D`\}$, point resource tickets $\{M,N\}$ and some of the spacial hashings (dotted lines) to the Chord, i.e., the d-dimensional coordinate values of a cell's control point is used as the DHT key and hashed on to the Chord ring. For this figure, f_{min} = 2, dim=2.

as the Chord method is utilized to hash these control points. So the responsibility for managing an index cell is associated with a Grid peer in the system. In Fig. 4.5, control point C is hashed to the Grid peer t, which is responsible for managing all claim and ticket objects that are stored with that control point (Claim X, Z and Ticket M).

For mapping claim objects, the process of mapping index cells to the Grid peers depends on whether it is a DPQ or DRQ. For a DPQ type query, the mapping is simple since every point is mapped to only one cell in the Cartesian space. For a DRQ type query, mapping is not always singular because a range lookup can cross more than one cell. To avoid mapping a DRQ to all the cells that it crosses (which can create many unnecessary duplicates), a mapping strategy based on diagonal hyperplane [55] of the Cartesian space is utilized. This mapping involves feeding a DRQ candidate index cell as an input into a mapping function, F_{map}. This function returns the IDs of index cells to which the given DRQ should be mapped. Spatial hashing is performed on these IDs (which returns keys for Chord space) to identify the current Grid peers responsible for managing the given keys. A Grid peer service uses the index cell(s) currently assigned to it and a set of known base index cells obtained at initialization as the candidate index cells.

Similarly, the mapping process of a ticket also involves the identification of the cell in

the Cartesian space. A ticket is always associated with a region [55] and all cells that fall fully or partially within that region will be selected to receive the corresponding ticket. The calculation of the region is based upon the diagonal hyperplane of the Cartesian space.

As discussed earlier in this section, a resource ticket encapsulates the number of processors available for a particular resource at a certain time. In this thesis, we assume that a claim object encapsulates the computational requirement for one processing unit. Thus, a resource ticket can match with one or more resource claim objects and in that case, all tasks corresponding to the matched claim objects are executed on the resource that issued the ticket object. Further, a claim object can also match with one or more ticket objects. In that case, the task corresponding to matched claim object is executed on the resource that issued the ticket object, which is at the head of ticket queue.

DHT Configured at Initialization

Similar to any P2P system (e.g. Gnutella [28]), we consider a Bootstrap Peer that computes the initial division (set of index cells that forms the basis for storing RUQs and RLQs) of indexing space based on configuration values, such as number of resource dimensions, maximum height of the distributed tree (f_{max}) and minimum division level (f_{min}). We assume that every other Grid Peer joining the overlay has access to IP address/DNS name of the Bootstrap peer. At initialization phase, a Grid Peer contacts the Bootstrap peer to obtain the list of index cells that it may be responsible for. The transfer of ownership of index cells (keys) is done based on DHT key management algorithm (e.g. Chord). To summarize, a Grid Peer is assigned those index cells whos ids lies between the Grid Peer's id and its predecessor's id. An example of Overlay creation, data indexing, object mapping and routing is illustrated in Fig. 4.6.

4.1.3 Application Model

We model scientific workflow application as Directed Acyclic Graph (DAG), where the tasks in the workflow are represented as nodes in the graph and the dependencies among the tasks are represented as the directed arcs among the nodes (see workflow model of Section 3.4 in Chapter 3). In general, a task in a workflow is a set of instructions that can

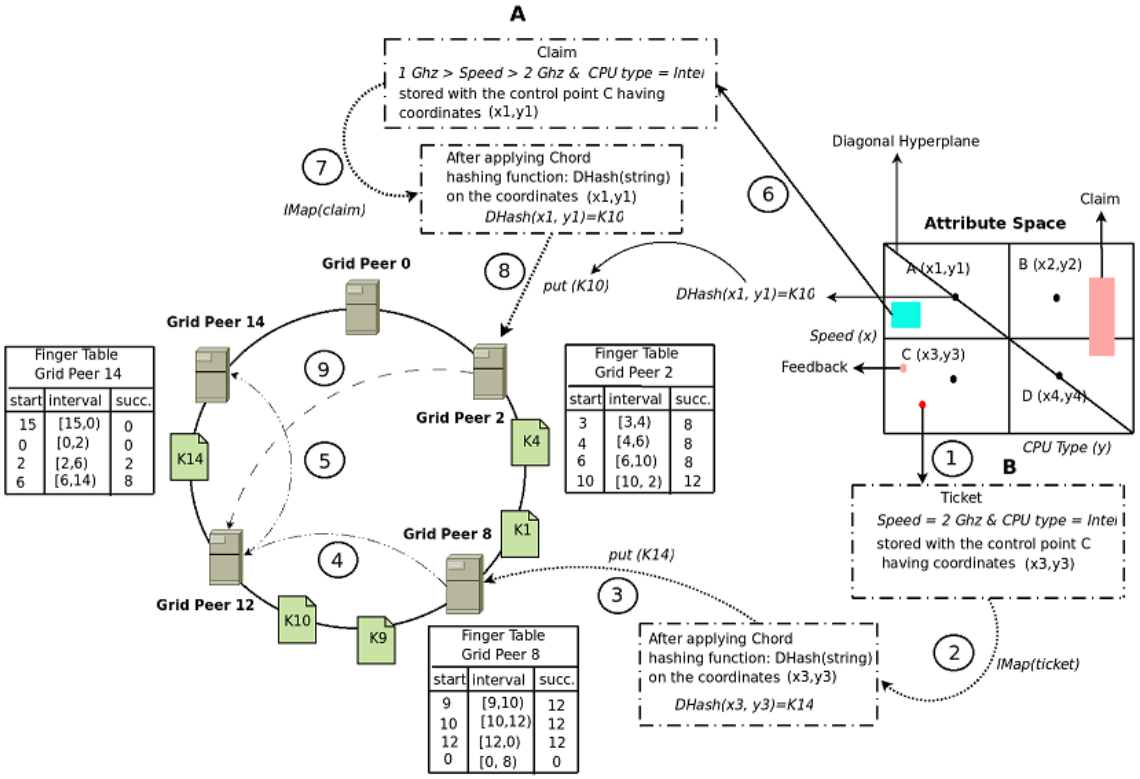


Figure 4.6: Overlay creation, data indexing, object mapping and routing: (1) a Grid site publishes ticket; (2) Grid peer 8 service computes the index cell, C($x3$,$y3$), to which the ticket maps by using mapping function $IMap$(ticket); (3) Next, distributed hashing function, DHash($x3$, $y3$), is applied on the cell's coordinate values, which yields a overlay key, $K14$; (4) Grid peer 8 based on its finger table entry forwards the request to peer 12; (5) Similarly, peer 12 on the overlay forwards the request to peer 14; (6) a GAS service submits a resource claim; (7) Grid peer 2 computes the index cell, C($x1$, $y1$), to which the claim maps; (8) DHash($x1$, $y1$) is applied that yields an overlay key, $K10$; (9) Grid peer 2 based on its finger table entry forwards the mapping request to peer 12.

be executed on a single processing element of a computing resource [24]. Examples of such workflow applications are [18], [120], [110], [32], and [96].

Definition 4.2 (Scientific Workflows): Scientific workflows describe a series of large number of structured activities and computations that arise in scientific problem solving. Usually, scientific workflows are data or computation intensive and the activities in the workflow have data or control dependencies among them.

Example 4.1: Let $V_{i,j,k}$ be the finite set of tasks $\{ T_1, T_2,...,T_x,...,T_y,T_m \}$ for the i-th submitted workflow from the j-th user of k-th workflow broker (GAS) and $E_{i,j,k}$ be the set of dependencies of the form $\{T_{x_{i,j,k}}, T_{y_{i,j,k}}\}$, where $T_{x_{i,j,k}}$ is the parent task of $T_{y_{i,j,k}}$. Thus, the i-th submitted workflow from the j-th user of k-th workflow broker in the system can be represented as,

$$W_{i,j,k} = \{V_{i,j,k}, E_{i,j,k}\}$$

In a workflow, we call a task that does not have any parent task, an entry task and a task that does not have any child task, an exit task. We also assume that a child task can not be executed until all of its parent tasks are completed. At any time of scheduling, the task that has all of its parent tasks finished is called a *ready* task.

4.2 Implementation

The architectural framework for autonomic workflow management system, presented in the previous section has been implemented by leveraging Aneka enterprise Grid platform [124] and following *Grid Federation* model. This section provides the implementation details (i.e. application environment, software components, and deployment scenario) of the proposed architecture.

4.2.1 Aneka Federation: An Overview

Aneka is a .NET-based service-oriented platform for constructing enterprise Grids. It is designed to support multiple application models, persistence and security solutions, and communication protocols such that the preferred selection of services can be changed at

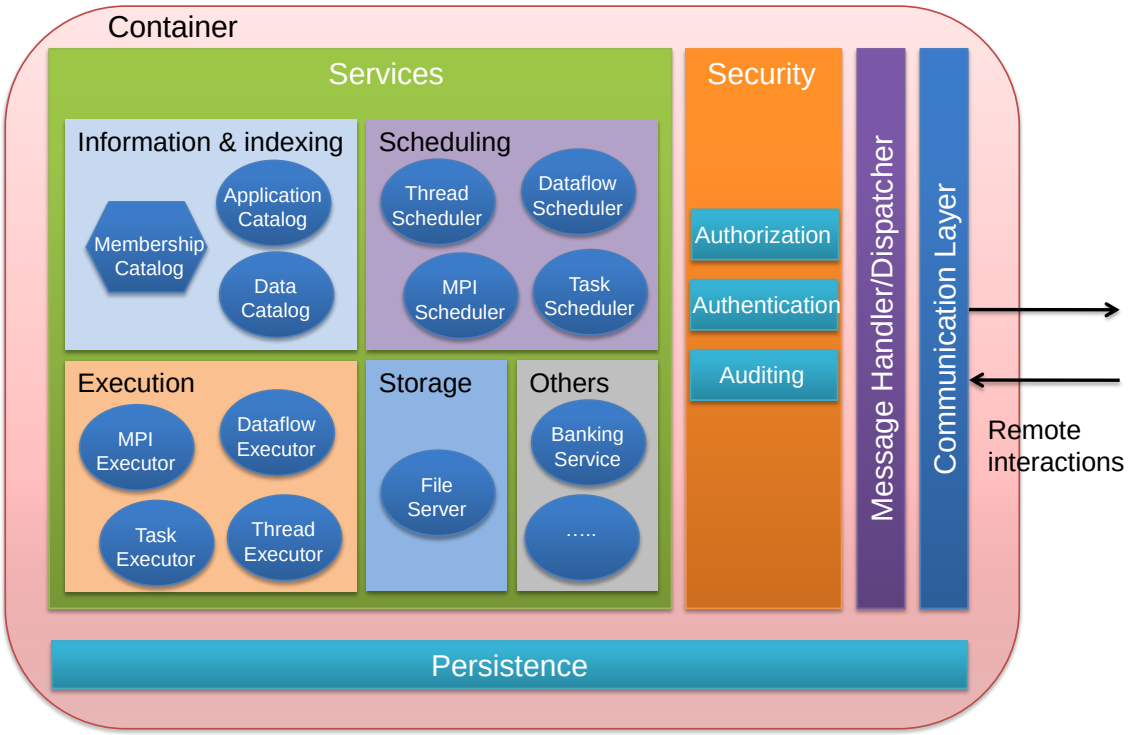


Figure 4.7: Aneka container.

anytime without affecting an existing Aneka ecosystem. To create an enterprise Grid, the resource provider only needs to start an instance of the configurable Aneka container (see Fig. 4.7) on each selected Grid node to host required services. The purpose of the Aneka container is to initialize services and act as a point for interaction with rest of the enterprise Grids.

The Aneka container can host any number of optional services that can be added to augment the capabilities of an enterprise Grid node. Examples of these optional services are indexing, scheduling, execution, and storage services. This provides a flexible and extensible framework for orchestrating different kinds of enterprise Grid application models. However, the Aneka container must host a MembershipCatalogue service that maintains the resource discovery indices (i.e. a .NET remoting address) of the services currently active in the system.

The Aneka Federation system logically connects topologically and administratively distributed Aneka enterprise Grid sites as part of a global resource sharing system. The components of Aneka Federation (computing resources and service providers) self-organize

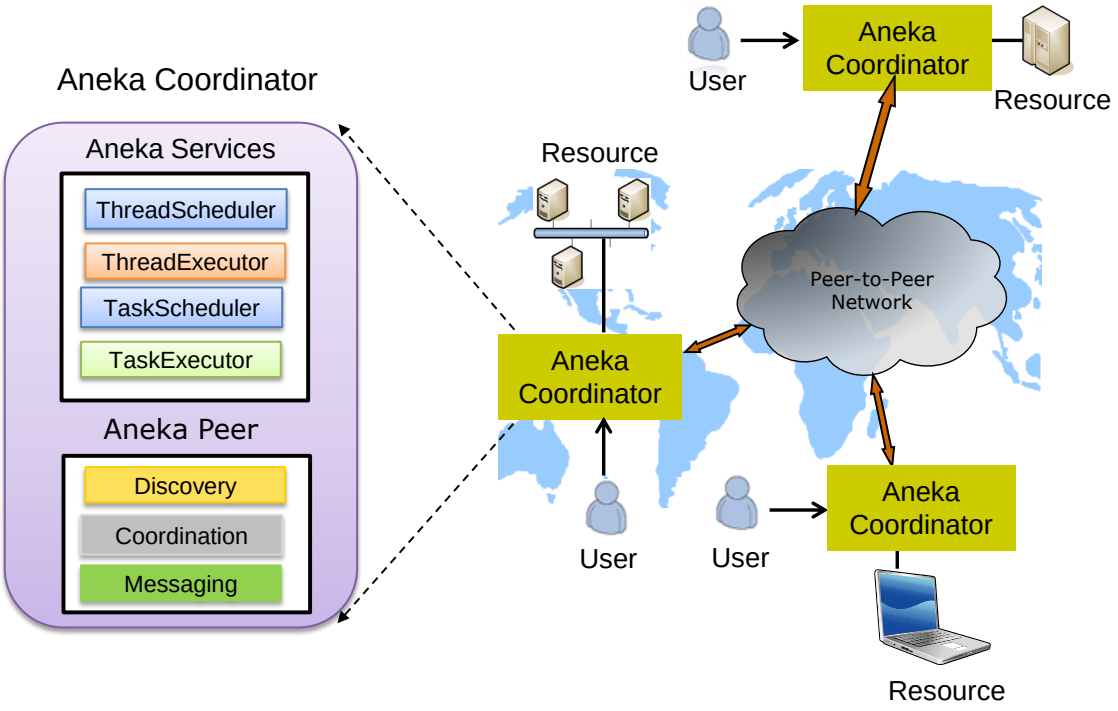


Figure 4.8: Aneka Federation network with the coordinator services.

themselves based on a DHT-based P2P routing methodology to create a scalable overlay of enterprise Grids. Each Grid site in the Aneka Federation instantiates a software service, called Aneka Coordinator. Depending on the scalability requirements and system size, an enterprise Grid can instantiate multiple Aneka Coordinator services. The Aneka Coordinator basically implements the functionalities for resource management and specifications for resource discovery protocol. The software design of the Aneka-Federation system decouples the fundamental decentralized interaction of participants from the resource allocation policies and the details of managing a specific Aneka Grid service. The Aneka Federation software sysbvtem utilizes the decentralized Grid services as regards to efficient distributed resource discovery and coordinated scheduling.

Components of Aneka Coordinator

Aneka Coordinator software service is mainly composed of two components: Aneka services and Aneka peer.

Aneka Services: Aneka Coordinator is responsible for managing a number of ser-

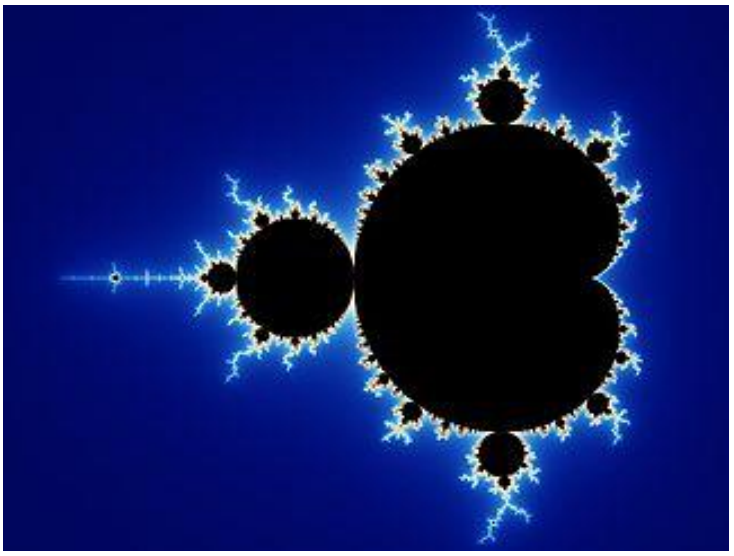


Figure 4.9: Mandelbrot application.

vices that include the core services for P2P scheduling (i.e. Thread Scheduler and Task Scheduler) and P2P execution (i.e. Thread Executor and Task Executor) provided by the Aneka framework. These services work independently in the container and have the ability to interact with other services, such as P2PMembershipCatalogue service through MessageDispatcher service deployed within each container.

Aneka Peer: It loosely joins the core Aneka services together with the decentralized Grid services (i.e. resource discovery, coordination, and messaging). Thus, the main functionality of Aneka peer is to provide services for content-based routing of lookup and update messages as well as facilitating decentralized coordination for efficient resource sharing and load-balancing among the distributed Aneka enterprise Grids.

Fig. 4.8 shows a distributed workflow management scenario, where an Aneka Coordinator is deployed at each enterprise Grid site and connected through a P2P overlay to create Aneka Federation.

4.2.2 Workflow Management in Aneka Federation

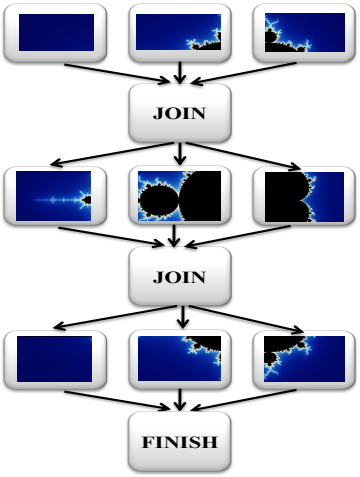
Application Development Environment

Application Model: Aneka supports composition and execution of applications based on two programming models: Task model and Thread model. The Task model defines an application as a collection of one or more tasks, where each task represents a unit of execution. Whereas, Thread model defines an application as a collection of one or more threads. To demonstrate the feasibility of implementing workflow management in Aneka Federation, we use Mandelbrot as an application.

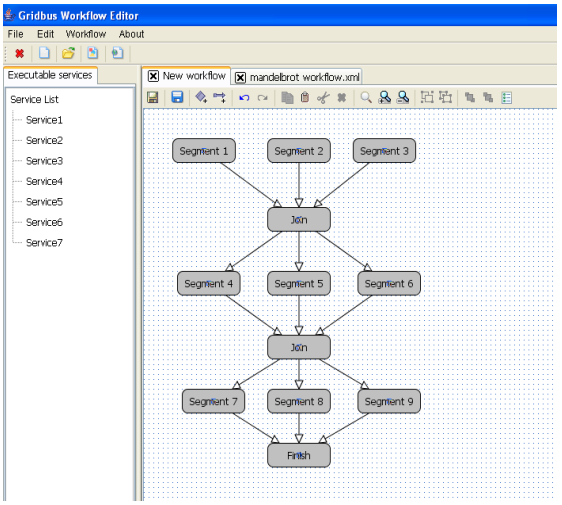
Mathematically, the Mandelbrot set is an ordered collection of points in the complex plane, the boundary of which forms a fractal. We enable the Mandelbrot fractal calculation in Grids using Task programming model. The application submission interface of Aneka platform allows the user to configure the number of horizontal and vertical partitions into which the fractal computation can be divided (see Fig. 4.10(c)). The number of tasks created is equal to *horizontal* x *vertical* partitions. For the demonstration purpose, This Mandelbrot application (see Fig. 4.9) is converted into a workflow as shown in Fig. 4.10(a). The representation of this workflow is then drawn in workflow editor (see Fig. 4.10(b)), which converts it into XML representation (see Fig. 4.10(d)). The Aneka application environment for Mandelbrot takes this XML definition of workflow as input and executes the tasks on Grid resources.

Workflow Editor: The workflow editor provides a Graphical User Interface (GUI) that enables users to create and modify existing workflows. The workflows are defined based on XML-based workflow language (xWFL), which has been used to model wide range of scientific workflow applications, such as Brain Image Analysis Workflow [86]. Using the editor, users design and create the workflows for complex scientific procedures by dragging and dropping boxes and arrows, connecting them, and defining their properties (input/output files, parameters etc.). Boxes and directed arrows represent workflow tasks and data dependencies between them, respectively.

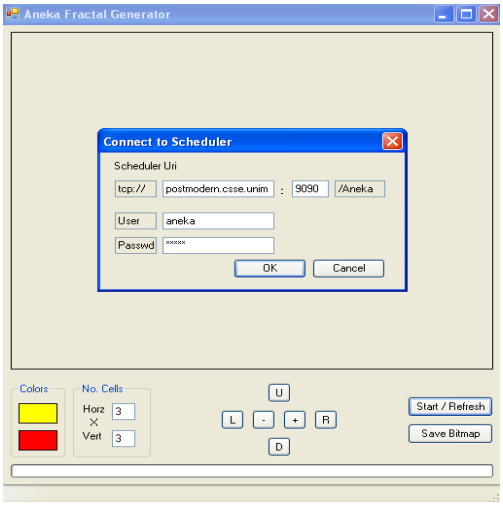
The GUI isolates users from the complexity of composing and editing XML entries in the files. However, for advance users, editor provides direct access to the XML details of the workflow design. Common editing operations, such as cut, copy, and paste can be



(a) Workflow application



(b) Visual representation of application in Workflow Editor



(c) Application execution interface

(d) XML representation of application in Workflow Editor

Figure 4.10: Mandelbrot application development and execution environment.

concurrently performed on multiple workflows. The workflow editor component is integrated with the Aneka application submission interface for facilitating users with regards to workflow composition and management.

Resource Discovery and Coordinated Scheduling

The resource discovery service is developed utilizing FreePastry P2P framework. FreePastry is an open source implementation of well-known Pastry routing substrate [107]. Pastry protocol was proposed by Microsoft's System Research Group, Cambridge, UK and Distributed System Group of Rice University. Pastry offers a scalable and efficient routing substrate for the development of P2P applications. The Aneka Coordinators create a Pastry overlay that collectively maintains a d-dimensional publish/subscribe index over the network of distributed coordinators as shown in Fig. 4.8 and facilitates decentralized resource discovery process.

The coordinated interaction among the resource providers and consumers in Aneka Federation is achieved through implementation of decentralized resource provisioning method. This method is engineered over a P2P routing and indexing system that has the ability to route, search, and manage complex coordination objects (i.e. claim and ticket) in the system. For scheduling tasks in a workflow, P2PScheduling service that extends Aneka's IndependentScheduling service implements the methods for (i) accepting workflow submission from client nodes; (ii) handling the task dependencies; (iii) sending search queries to P2PMembershipCatalogue service; (iv) dispatching tasks to P2PExecution services after mapping; and (v) collecting the execution output. The interaction between P2PScheduling and P2PExecution services is facilitated by the MembershipCatalogue service as it accepts resource ticket objects from P2PExecution service and resource claim objects from P2PScheduling service through MessageDispatcher.

Deployment

To realize Aneka Coordinator software service, Aneka peer component is implemented using C#.Net, while the publish/subscribe indexing component is implemented using Java. Thus, in order to facilitate inter-operation between these two components, web service interfaces are implemented for both components. The P2PMembershipCatalogue of Aneka

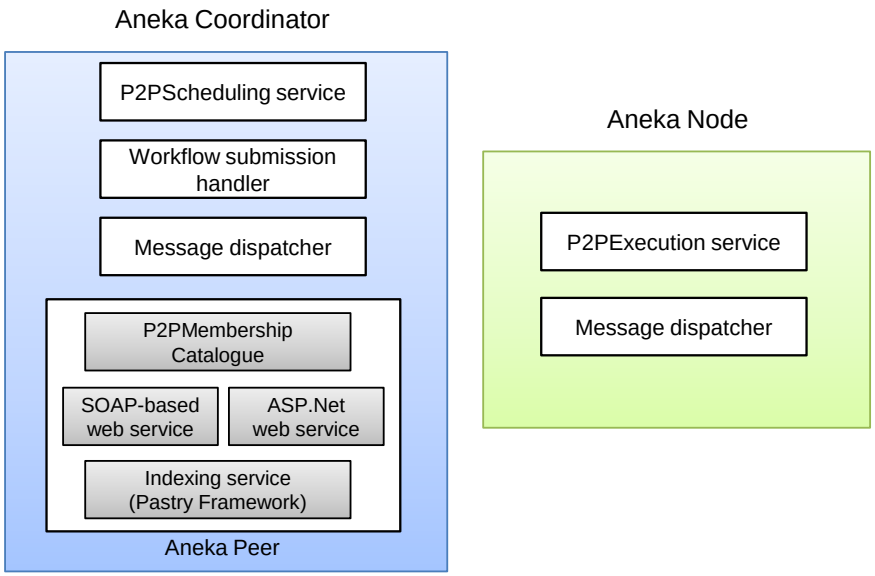


Figure 4.11: Various services hosted by Aneka Coordinator and Aneka Execution nodes.

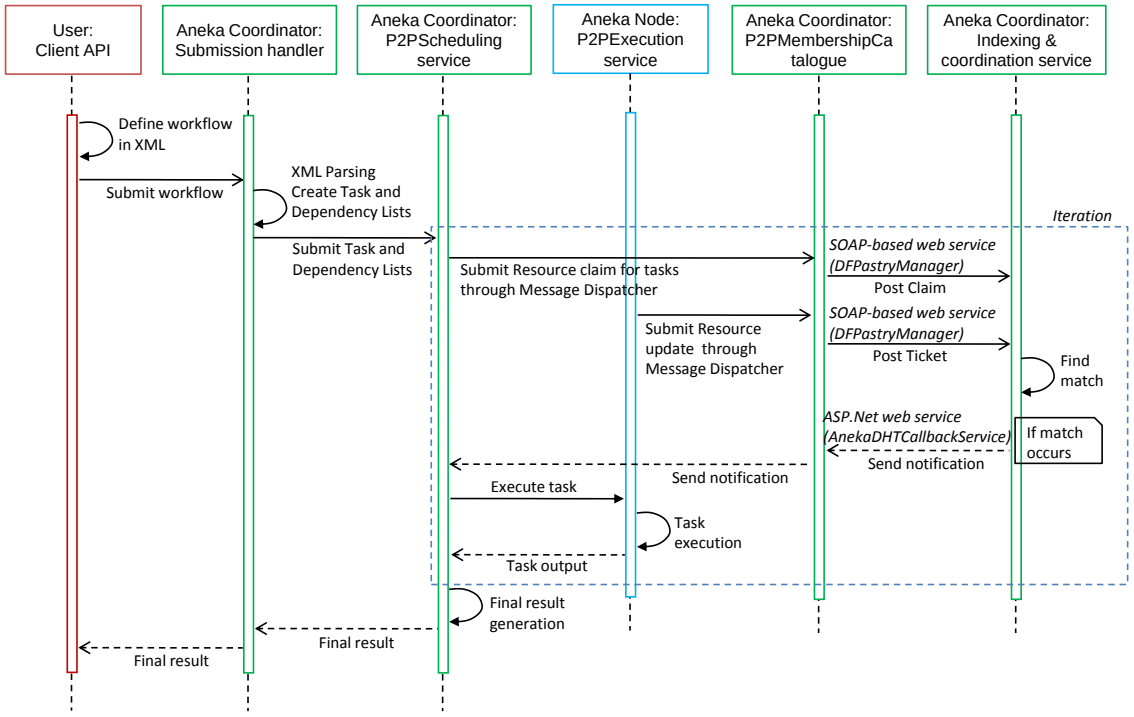


Figure 4.12: Sequence diagram of the interaction among different Aneka components for the lifecycle of a workflow execution.

peer service interacts with the publish/subscribe indexing service hosted in Apache Tomcat container in order to post the resource tickets and claims, which is achieved using Simple Object Access Protocol (SOAP) based web service. On the other hand, to receive query responses from the publish/subscribe indexing component, a .Net web service is implemented using ASP.Net and hosted by Microsoft Internet Information Service 6.0 (IIS) container. The communication messages are specified in XML and parsed by utilizing the Apache Axis 1.4 SOAP engine. Similarly, workflow definition is also specified in XML and parsed by the P2PScheduling service of Aneka Coordinator for mapping the tasks to resources and taking scheduling decisions. The services hosted by Aneka Coordinator and Aneka Execution nodes in order to facilitate execution of workflows in Aneka Federation are presented in Fig. 4.11. In addition, Fig. 4.12 illustrates the interaction among these services throughout the lifecycle of a workflow execution.

A sample deployment scenario of autonomic workflow management system in Aneka Federation is depicted in Fig. 4.13, where four Aneka enterprise/desktop Grids are connected to form an Aneka Federation. In this deployment scenario, one node of each Grid site instantiates the Aneka Coordinator services including P2PScheduling service and is responsible for discovering and coordinating the provisioning of distributed resources in Aneka Federation, whereas the other nodes instantiate only the P2PExecution service. Scientists and users utilize the Aneka application submission interface for submitting workflow applications to the local Aneka Coordinator service. The P2PExecution services periodically update their availability and usage status to the Aneka Coordinators and the P2PScheduling services of these coordinators perform decentralized task scheduling and resource provisioning on the self-managing P2P overlay.

4.3 Conclusion

This chapter presents an architectural framework for autonomic workflow management system for Grid computing environment. In particular, we describe the system models, such as Grid model, coordination model, and application model that form the basis of this architectural framework. Moreover, the prototype implementation of the proposed architecture is also discussed in this chapter. The prototype wokflow management system

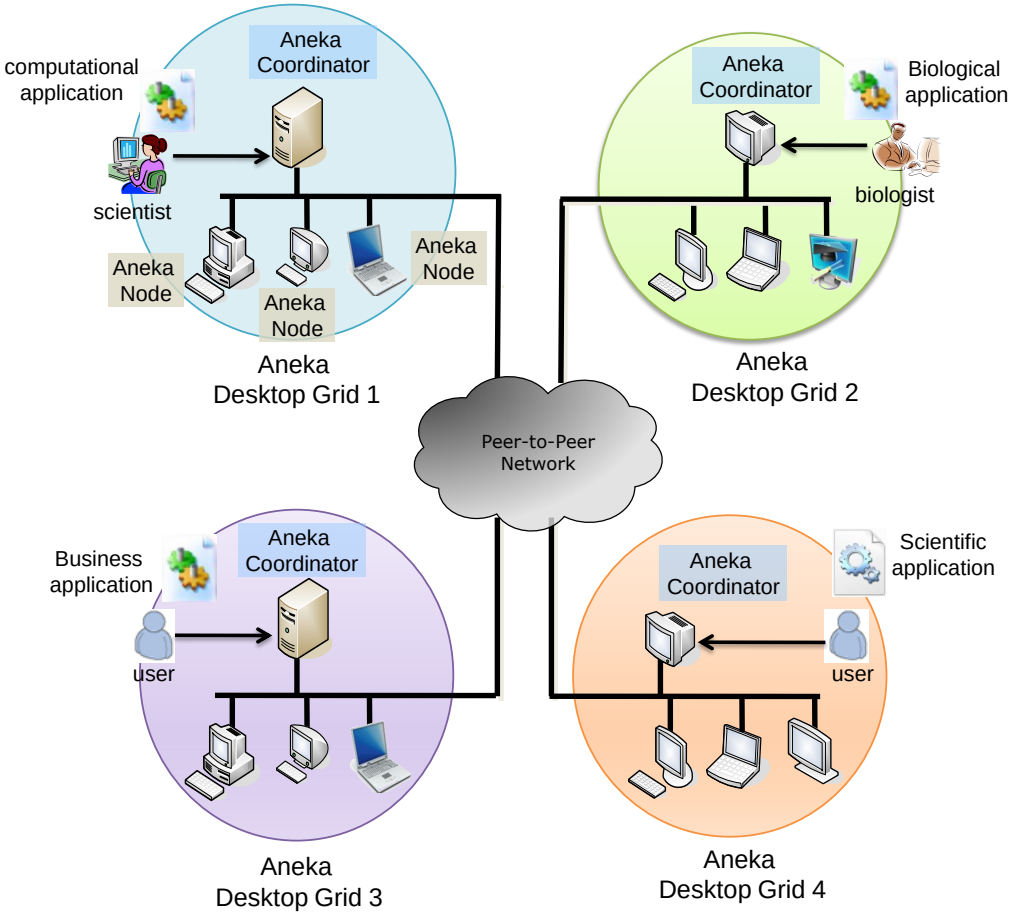


Figure 4.13: Workflow execution scenario in Aneka Federation.

is developed by leveraging Aneka enterprise Grid platform that enables the creation of Aneka Federation, a model for decentralized resource sharing and application scheduling. In addition, the required components and services of the prototype system, such as application development, resource discovery, coordination, and task scheduling are also illustrated with the direction for deployment of the system. The workflow management architecture presented in this chapter is self-configuring in nature and further utilized in the next two chapters for developing self-optimizing and self-healing workflow management policies for Grids.

Chapter 5

Decentralized and Cooperative Workflow Scheduling

This chapter presents a decentralized and cooperative scheduling technique utilizing a self-configuring P2P overlay for workflow applications in global Grids based on the workflow management system architecture discussed in Chapter 4. Using simulation, the performance of this scheduling approach is measured with respect to scheduling and coordination perspectives. The results show that our approach is scalable in terms of scheduling message complexity and able to optimize the execution completion time. As we leverage the DHT-based coordination space for scheduling decision making, it can also avoid the limitation of single point of failure with regards to resource information service in centralized scheduling techniques.

5.1 Introduction

Workflow scheduling is a process of finding the efficient mapping of tasks in a workflow to the suitable resources so that the execution can be completed with the satisfaction of objective functions, such as execution time minimization as specified by Grid users. Therefore, the efficiency of the workflow scheduling algorithm directly affects the performance of Grid systems with respect to delivered Quality of Service (QoS) and resource utilization. Majority of the existing workflow scheduling approaches are non-coordinated

(refer to Section 5.5), where workflow schedulers perform scheduling related activities independent of the other schedulers in the system. They directly submit their applications to the underlying Grid resources without taking into account the current load, priorities, and utilization scenarios of other application level schedulers, which leads to over-utilization or a bottleneck on some valuable resources, while leaving others largely under-utilized. Further, these brokers do not have a coordination mechanism, and this exacerbates the load sharing and utilization problems of distributed Grid resources.

Moreover, current brokering systems have evolved around centralized client/server or hierarchical models, where the responsibilities of key functionalities, such as resource discovery are delegated to the centralized server machines. Centralized models have well-known drawbacks regarding scalability, single point of failure, and network congestion at links leading to the server.

To overcome these problems, this chapter proposes a novel approach for decentralized and cooperative workflow scheduling in a dynamic and distributed Grid resource sharing environment. The proposed approach derives from a Distributed Hash Table (DHT)-based coordination space that provides a global virtual shared with regards to resource discovery, coordination and overall system decentralization. New generation DHT-based routing algorithms, such as Chord [115] form the basis for organizing the coordination space and creating the structured P2P overlay to arrange the Grid resources. The self-organizing and decentralized nature of P2P overlay facilitates automatic and seamless configuration of the resource organization in the event of resource failure, join, or leave. Furthermore, the process of cooperative decision making for scheduling ensures the optimization of workflow execution considering the dynamic resource behaviour.

In a nutshell, in this chapter, we present a task scheduling algorithm for determining task dependency and priority, a resource provisioning algorithm for mapping these tasks to suitable Grid resources, and a resource coordination algorithm for coordinating interactions among the distributed entities in the system. Next, we perform a comprehensive simulation based study of the proposed scheduling approach and analyze different performance metrics to demonstrate its performance and efficiency. Moreover, we also evaluate our approach against non-cooperative scheduling and centralized coordination techniques with respect to two performance dimensions, load balancing and coordination efficiency.

Table 5.1: Notations: resource, workflow, and coordination models

Symbol	Meaning
Resource	
r	number of resources or GASs in the Grid system
R_i	i-th Resource in the system
G_i	i-th GAS in the system
t_p	number of task executions per processor
Workflow	
t	number of tasks in a workflow.
T	total number of tasks in the system.
e	total number of inter-task dependencies in a workflow.
l	number of levels in a workflow.
w	width of a fork-join workflow.
α	number of tasks in a level that can be executed concurrently.
$W_{i,j,k}$	i-th workflow from the j-th user of k-th GAS in the system.
$T_{x_{i,j,k}}$	x-th task of the workflow, $W_{i,j,k}$.
Coordination	
$r_{x_{i,j,k}}$	a claim posted for task $T_{x_{i,j,k}}$.
u_i	a ticket issued by the i-th GAS/broker.
dim	dimensionality or number of attributes in the Cartesian space.
f_{min}	minimum division level of d-dimensional index tree.

Table 5.2: Notations: scheduling and network models

Symbol	Meaning
Scheduling	
$\overline{CD}$	average coordination delay per task in the system.
$\overline{RT}$	average response time per task.
$\overline{MS}$	average makespan per workflow.
NT	total number of notifications for all tasks.
HP	average number of routing hops per claim/ticket.
CL	total number of claims in the system.
$E(T_{x_{i,j,k}})$	execution time for task $T_{x_{i,j,k}}$.
$D(T_{(xy)_{i,j,k}})$	data transfer time from task $T_{x_{i,j,k}}$ to $T_{y_{i,j,k}}$.
$rank(T_{x_{i,j,k}})$	rank value of task $T_{x_{i,j,k}}$.
Network	
λ^{in}	total incoming rate a Chord service from the network queue.
λ^{out}	outgoing claim/ticket rate at a network queue.
μ_n	average network queue service rate at a Grid peer.
μ_r	average query reply rate for index service at GFA.
λ_t^{in}	incoming ticket rate at a index service.
λ_c^{in}	incoming claim rate at a index service.
λ_a^{in}	incoming query rate at a DHT routing service from the local index service.
λ_{index}^{in}	incoming index query rate at a index service from its local DHT routing service.
K	network queue size .

With the implementation of this approach, not only the performance bottlenecks are eliminated but also efficient scheduling with enhanced scalability is achieved.

5.2 System Models

The proposed scheduling algorithm utilizes the Grid Federation [101] model discussed in Chapter 4 with regards to distributed resource organization and Grid networking.

We also consider the Scientific workflow applications as the case study for the proposed scheduling approach. The modeling of this kind of applications is described in Section 4.1.3 of Chapter 4.

5.3 Proposed Algorithms

In this section, we provide description of the algorithms that have been developed for: (i) task scheduling; (ii) resource provisioning; and (iii) resource coordination. In Table 5.2, we show the model parameters related to scheduling and P2P coordination network.

5.3.1 Scheduling and Provisioning Algorithm

Task Scheduling

Here, we discuss about the task scheduling algorithm (refer to Algorithm 2) that is undertaken by a GAS in the Grid Federation system on the arrival of a job or workflow. When a user submits a workflow application, $W_{i,j,k}$ to a GAS, the GAS calculates the priority of each task (line 12-15). Earliest Finish Time (EFT) heuristic is used to calculate task priorities. This heuristic first computes the execution time for each task and communication time between resources of two successive tasks in the workflow (line 1-9). The execution time of a task on a computing resource is calculated by dividing the size of task (e.g. Million Instructions) by CPU capability of the corresponding resource (e.g. Million Instructions Per Second). Let $|T_{x_{i,j,k}}|$ be the size of task $T_{x_{i,j,k}}$, submitted to G_i and R_i be the local resource with the processing power, $|R_i|$. Thus, the execution time of the task is defined as,

$$E(T_{x_{i,j,k}}) = \frac{|T_{x_{i,j,k}}|}{|R_i|} \tag{5.1}$$

Let $\overline{T}_{(xy)_{i,j,k}}$ be the size of data to be transferred between task $T_{x_{i,j,k}}$ and $T_{y_{i,j,k}}$, submitted to G_i and R_i be the local resource with the data processing capacity, $\overline{R}_i$. Thus, the

data transfer time for the task is defined as,

$$D(T_{xy_{i,j,k}}) = \frac{\overline{T}_{(xy)_{i,j,k}}}{\overline{R}_i} \tag{5.2}$$

$E(T_{x_{i,j,k}})$ and $D(T_{(xy)_{i,j,k}})$ are used to calculate the rank of a task. For an exit task, the rank value is,

$$rank(T_{x_{i,j,k}}) = E(T_{x_{i,j,k}}) \tag{5.3}$$

Now, the rank value of other tasks in the workflow can be computed recursively based on equations (5.1), (5.2), and (5.3) and is represented as,

Algorithm 2 TASK SCHEDULING AT GAS

PROCEDURE: Initialize Task Priority
Input: Workflow W_{ijk}
begin
  for all $t \in TaskList$ of workflow W_{ijk}
    Calculate execution time for t according to (5.1)
  end for
  for all $e \in EdgeList$ of workflow W_{ijk}
    Calculate data transfer time for e according to (5.2)
  end for
  Run BFS following reverse task dependency and calculate rank value for each task according to (5.3) (5.4)
end
PROCEDURE: Event User Workflow Submit
Input: Workflow W_{ijk}
begin
  Initialize Task Priority (W_{ijk})
  Generate *Ready TaskList* for W_{ijk}
  Submit *Ready* tasks for execution
end
PROCEDURE: Event Task Finish Notification
Input: Task $T_{x_{ijk}}$
begin
  Update dependency list of each task in $TaskList$
  Generate *Ready TaskList* for W_{ijk}
  Submit *Ready* tasks for execution
end

$$rank(T_{x_{i,j,k}}) = \max_{T_{y_{i,j,k}} \in succ(T_{x_{i,j,k}})} (D(T_{(xy)_{i,j,k}}) + rank(T_{y_{i,j,k}})) + E(T_{x_{i,j,k}}) \quad (5.4)$$

Since a workflow is represented as a DAG, the rank values of the tasks are calculated (line 10) by traversing the task graph in a Breadth First Search (BFS) manner in the reverse direction of task dependencies (i.e. starting from the exit tasks).

Once the rank values are calculated, the GAS generates the 'ready' tasks in the $TaskList$ based on the dependency of each task and put them into the $ReadyTaskList$ (line 16). Finally, the GAS submits the 'ready' tasks for execution (line 17). Further, when the GAS receives a notification message stating task, $T_{x_{i,j,k}}$ has finished execution, it first updates the dependency lists of the tasks that are dependant on $T_{x_{i,j,k}}$ (line 19-22); then it computes the 'ready' tasks at that moment and submits them for execution (line 23-24).

Resource Provisioning

The details of the decentralized resource provisioning algorithm (refer to Algorithm 3) that is undertaken by the P2P coordination space is presented here. When a resource claim object, $r_{x_{ijk}}$ arrives at the coordination service, it is added to the existing claim list, *ClaimList* by the coordination service (line 16-20). When a resource ticket object, u_i arrives at the coordination service, the list of resource claims that overlap or match with the submitted resource ticket object in the d-dimensional space is computed (line 21-25). The overlap signifies that the task associated with the given claim object can be executed on the ticket issuer's resource subject to its availability.

In order to get the matches, the coordination service first, sorts the claim objects in the *ClaimList* in descending order according to their rank value (line 4-5); then from this list, the number of claims that overlap with the ticket are selected to the $ClaimList_m$ (line 6-13). From the $ClaimList_m$, the resource claimers are selected one by one based on their rank value(higher rank first) and the resource claimers are notified about the resource ticket match until the ticket issuer is not over-provisioned (line 25-30). The coordination procedure can utilize the dynamic resource parameters, such as the number of available processors, queue length etc. as the over-provision indicator. These over-provision indicators are encapsulated with the resource ticket object by the GASs.

5.3.2 Coordination Algorithm

This section provides the description of the algorithm (refer to Algorithm 4) for coordinating the interactions among different entities in the system. When a GAS, G_j intends to submit a task, $T_{x_{ijk}}$ for execution, it compiles a resource claim object, $r_{x_{ijk}}$ for the task and posts it to the P2P coordination space (line 1-6). On the other hand, the GAS, G_i compiles the resource ticket object, u_i for resource, R_i and publishes it to the P2P coordination space periodically depending on the ticket injection rate (refer to Section 5.4.2).

Algorithm 3 RESOURCE PROVISIONING AT COORDINATION SPACE

1: **PROCEDURE**: Match
2: Input: Ticket u_i from Resource R_i
3: **begin**
4: Obtain rank value of each task in the ***ClaimList***
5: Sort ***ClaimList*** in descending order of task's rank value
6: $index \leftarrow 0$
7: $ClaimList_m \leftarrow \Phi$
8: **while** ***ClaimList***[index] $\neq null$ **do**
9: $r_{x_{ijk}} \leftarrow$ ***ClaimList***[index]
10: **if** $r_{x_{ijk}} \cap u_i \neq null$ **then**
11: $ClaimList_m \leftarrow ClaimList_m \cup r_{x_{ijk}}$
12: **end if**
13: $index \leftarrow index + 1$
14: **end**
15: **return** $ClaimList_m$
16: **end**
17: **PROCEDURE**: Event Resource Claim Submit
18: Input: Claim $r_{x_{ijk}}$
19: **begin**
20: $ClaimList \leftarrow ClaimList \cup r_{x_{ijk}}$
21: **end**
22: **PROCEDURE**: Event Resource Ticket Submit
23: Input: Ticket u_i from Resource R_i
24: **begin**
25: $ClaimList_m \leftarrow$ Match(u_i)
26: $index \leftarrow 0$
27: **while** R_i is not over-provisioned **do**
28: Send notification of match event to resource claimer $ClaimList_m$[index]
29: Remove $ClaimList_m$[index]
30: $index \leftarrow index + 1$
31: **end**
32: **end**

Besides, whenever the resource condition changes, such as a task completion event happens, the GAS also posts a ticket object for the corresponding resource immediately (line 7-12).

Once there is a match between a ticket object, u_i and a claim object, $r_{x_{ijk}}$, the coor-

Algorithm 4 RESOURCE COORDINATION

1: **PROCEDURE**: Event Task Submit
2: Input: Task T_{ijk}
3: **begin**
4: Encapsulate claim object $r_{x_{ijk}}$ for task $T_{x_{ijk}}$
5: Subscribe $r_{x_{ijk}}$ to coordination space
6: **end**
7: **PROCEDURE**: Event Resource Status Changed
8: Input: Resource R_i
9: **begin**
10: Encapsulate ticket object u_i for resource R_i
11: Publish u_i to coordination space
12: **end**
13: **PROCEDURE**: Event Ticket Redeemption Request
14: Input: Task $T_{x_{ijk}}$, GAS G_i
15: **begin**
16: Send ticket redeemption *Request* for task $T_{x_{ijk}}$ to GAS G_i; *Request* message implies G_j
17: **end**
18: **PROCEDURE**: Event Ticket Redeemption Reply
19: Input: Task $T_{x_{ijk}}$, GAS G_j
20: **begin**
21: **if** $T_{x_{ijk}}$ can be executed on local resource **then**
22: Send *Reply* "accept" to GAS G_j
23: **else**
24: Send *Reply* "reject" to GAS G_j
25: **endif**
26: **end**
27: **PROCEDURE**: Event Ticket Redeemption Reply Action
28: Input: Task $T_{x_{ijk}}$, GAS G_i, *Reply*
29: **begin**
30: **if** *Reply* is "accept" **then**
31: Send $T_{x_{ijk}}$ to accepting GAS G_i for execution
32: Unsubscribes the claim object for $T_{x_{ijk}}$
33: **else**
34: Subscribe claim object $r_{x_{ijk}}$ to coordination space
35: **endif**
36: **end**

dination service sends a ticket redeemption request for task, $T_{x_{ijk}}$ to ticket issuer GAS, G_i (line 13-17). The request message contains the information that task, $T_{x_{ijk}}$ is submitted by GAS, G_j. After notifying the resource claimer GAS, the coordination service unsubscribes the resource claim for that task from the P2P coordination space.

When the ticket issuer GAS, G_i receives the notification of match from the coordination service, it sends a $Reply$ to the resource claimer GAS, G_j. If the task can be executed on the local resource at that time, it sends a positive reply; otherwise it sends a negetive reply to the claimer GAS (line 18-26). If the $Reply$ is 'accept', then the claimer GAS, G_j transfers the locally submitted task, $T_{x_{ijk}}$ to the ticket issuer GAS and unsubscribes the claim object from the coordination space to remove duplicates (line 27-32). However, if the ticket issuer GAS fails to grant access due to local resource sharing policy (i.e. $Reply$ is 'reject'), then the claimer GAS reposts the resource claim object for that task to the coordination space for future notifications (line 33-34). The interaction between different entities in the system is depicted in Fig. 5.1.

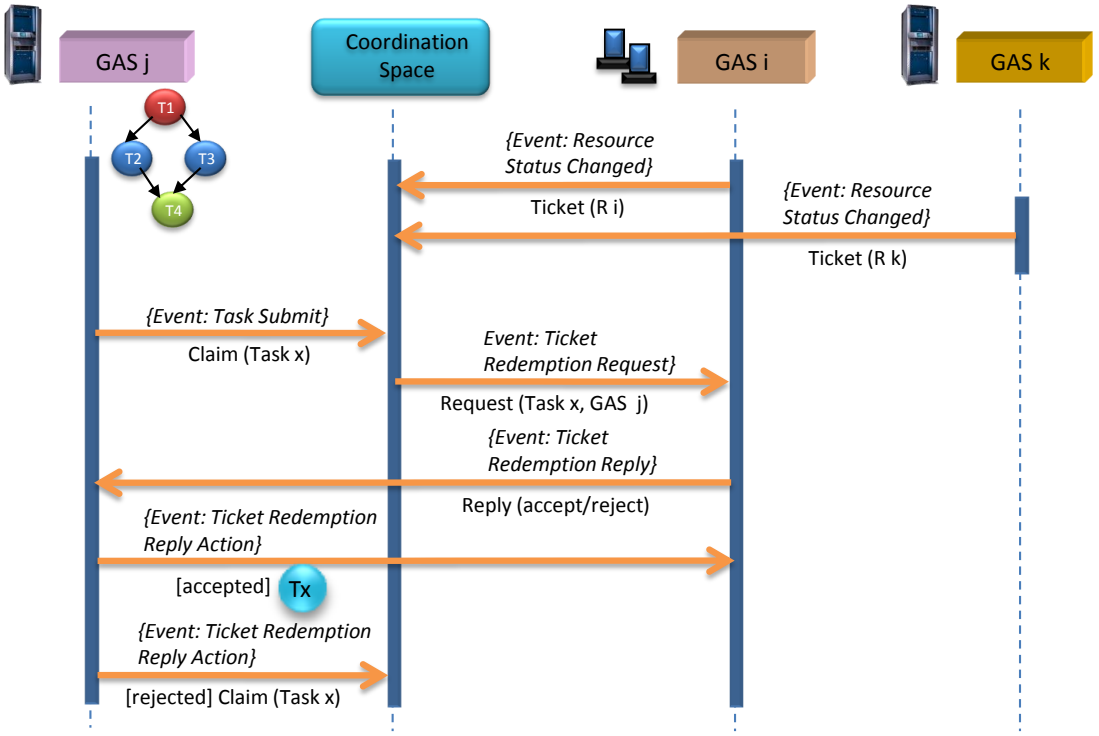


Figure 5.1: Interaction between various entities in the system during resource coordination.

5.3.3 Time Complexity

Let us consider r number of Grid resources. Thus, there are r number of GASs in the system. Let each user associated with a GAS submits a workflow consisting of t number of tasks and e number of dependencies among the tasks. Then the complexity of calculating execution times of the tasks in the workflow is $O(t)$ and data transfer times for the dependencies in the workflow is $O(e)$. Further, the complexity of running BFS to compute the rank values of all the tasks is $O(e + t)$ and if an adjacency list is used to handle the dependencies, then the complexity of generating 'ready' tasks is $O(e)$. Therefore, the overall time complexity of Algorithm 2 is $O(e + t)$.

In worst case, $ClaimList$ in the Algorithm 3 can contain $r.t$ number of tasks. So the complexity of sorting the $ClaimList$ is $O((r.t)\log(r.t))$ and finding out the total number of matches is $O(r.t)$. Calculating the number of resource claimers to be notified about the matches also requires $O(r.t)$ steps in worst case. Thus, the overall complexity of Resource Provisioning algorithm is $O((r.t)\log(r.t))$.

The time complexity of resource coordination algorithm is $O(1)$.

5.3.4 Example

An example scenario of the process of task scheduling, resource provisioning and coordination according to the proposed approach is illustrated in Fig. 5.2. The steps are as follows:

1. $user1$ submits the workflow, $W1$ to the local broker, $GAS1$.
2. $GAS1$ ranks the tasks in the workflow using EFT heuristic.
3. $GAS1$ picks the ready task with higher rank ($T1$) and posts or subscribes a resource claim to the coordinatin space.
4. All the GASs in the system sends or publish the resource ticket or Resource Update Query (RUQ) to the P2P coordination space.
5. In the P2P coordination space, resource ticket of $GAS3$ matches with the resource claim of $GAS1$.

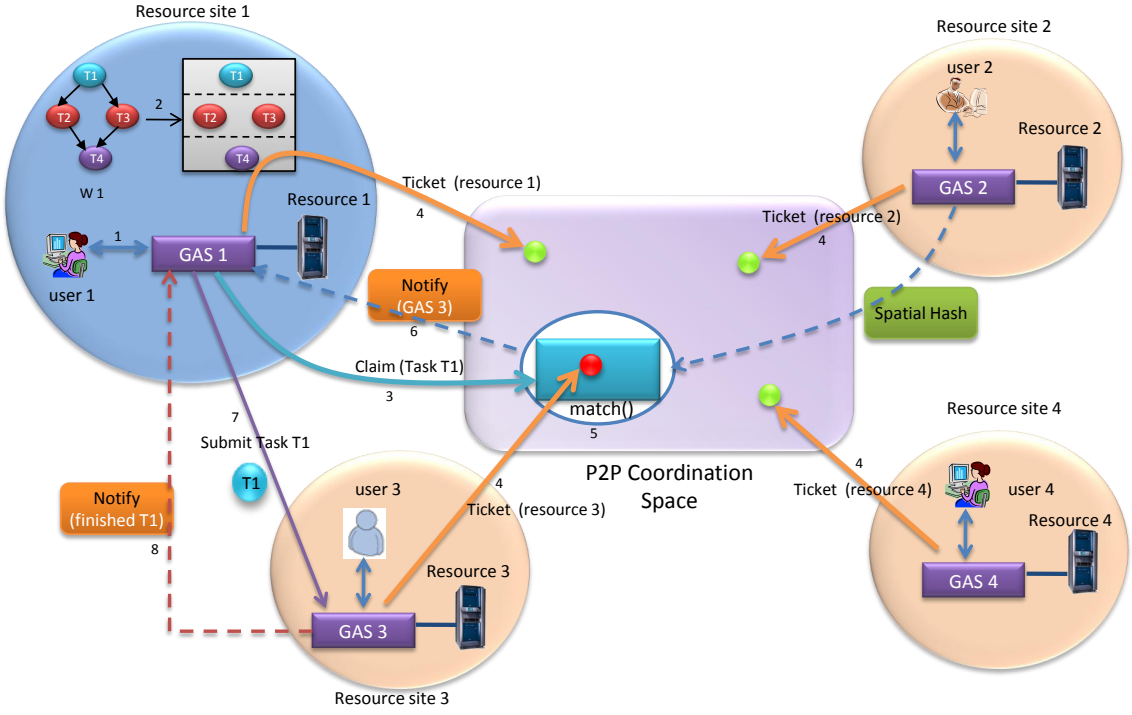


Figure 5.2: Example of the process of task scheduling, resource provisioning, and coordination.

6. $GAS1$ receives the notification message of resource matching ($GAS3$) from the coordination space.

7. $GAS1$ submits task $T1$ to $GAS3$.

8. $T1$ is executed on $Resource3$ and $GAS3$ notifies $GAS1$ of task completion; steps 4-8 continues until all the tasks in $W1$ have been finished execution.

5.4 Performance Evaluation

5.4.1 Network Model

The network model, considered for the simulation study is an interconnected network of r Grid peers, where a Grid peer node (through its Chord routing service) is connected to an network message queue and an incoming link from the Internet (as shown in Fig. 5.3). The network message queue accepts the following two types of incoming messages from:

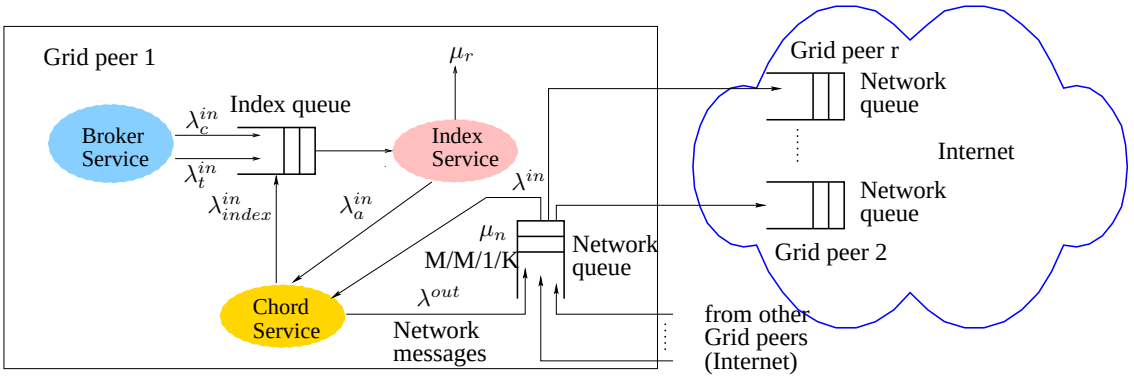


Figure 5.3: Network message queueing model at a Grid peer.

(i) other Grid peers on the Chord overlay; and (ii) the local Chord service. The Chord service receives messages from local publish/subscribe index service (at rate λ_a^{in}) and the network message queue (at rate λ^{in}). These messages are processed as soon as they arrive at the Chord routing service. After processing, Chord routing service queues the messages in the local network message queue at a rate, λ^{out}. We denote the message processing rate of network message queue by μ_n. Basically, this queue models the network latencies that a message encounters as it is transferred from one Chord routing service to another on the Grid Federation overlay. The distributions for the delays (including queuing and processing) encountered in a network message queue are given by the M/M/1/K queue steady state probabilities.

We denote the rates for claim and ticket object by λ_c^{in} and λ_t^{in} respectively. The queries are directly sent to the local index service, which first processes them and then forwards them to the local Chord routing service. Although, we consider a message queue for the index service but we do not take into account the queuing and processing delays as it is in microseconds. Index service also receives messages from the Chord routing service at a rate λ_{index}^{in}. The index messages include the claims and tickets that map to the control area currently owned by the Grid peer, and the notification messages arriving from the network. The local index service sends message to its Chord service at a rate, λ_a^{in}. The index service sends notification messages (claim-ticket match message) to its broker service at a rate, μ_r.

5.4.2 Simulation Setup

Our simulation infrastructure is created by combining two discrete event simulators namely *GridSim* [22], and *PlanetSim* [49]. GridSim offers a concrete base framework for simulation of different kinds of heterogeneous resources, services and application types. PlanetSim is an event-based overlay network simulator that can simulate both unstructured and structured P2P overlay networks.

Workload Configuration

We utilize the workflow generator discussed in Chapter 3 for creating various formats of weighted pseudo-application workflows. The following input parameters are used to create a workflow.

- t, the total number of tasks in the workflow.
- l, the total number of levels in the workflow. l represents the ratio of the total number of tasks to the width (i.e. maximum number of tasks in a level). Hence, width: w= $\frac{t-1-\frac{l-1}{2}}{\frac{l-1}{2}}$.

In this study, we consider fork-join workflow (refer to Fig. 5.4) and an example of such workflow is WIEN2K [18], which is a quantum chemistry application developed at Vienna University of Technology. In this kind of workflow, forks of tasks are created and then joined, such that there can be only one entry task and one exit task. But the number of tasks at each level depends on total number of tasks and the width of that level, w. We vary the number of tasks in a workflow over the interval $[50, 500]$ and the size of each task is randomly generated form a uniform distribution between 50000 MI (Million Instruction) to 500000 MI. Further, we assume that workflows are computation intensive. Thus, the data dependency among the tasks in the workflow is negligible.

Network Configuration

The experiments run a Chord overlay with 32 bit configuration, i.e. number of bits utilized to generate node and key ids. The GAS/broker network size r is fixed to 100. Further,

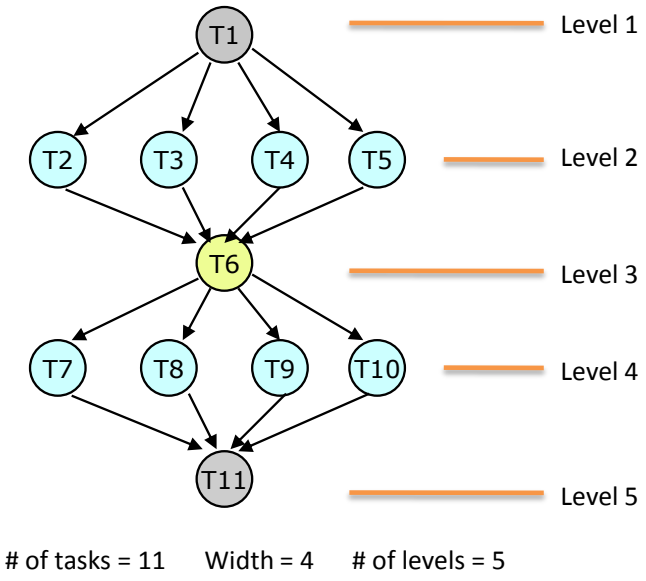


Figure 5.4: A fork-join workflow.

network queue message processing rate, μ_n, is fixed at 4000 messages per second and message queue size, K is fixed at 10^4.

Resource Claim and Ticket Injection Rate

The GASs inject the ticket objects based on the exponential inter-arrival time distribution. The injection rate (i.e. RUQ rate), λ_t^{in}, for the resource tickets is distributed over the interval $[100, 300]$ in step of 100 secs. Note that, the inter-arrival delay between injecting the ticket objects is modeled to be the same for all the GASs in the system. At the beginning of the simulation, the resource claims for the entry tasks of all the workflows in the system are injected. Subsequently, when these tasks finish, then the resource claims for the successive tasks in the workflow are posted. This process is repeated until all the tasks in the workflow are successfully completed. Spatial extent of both resource claims and ticket objects lie in a 4-dimensional attribute space. These attribute dimensions include the number of processors, their speed, their architecture, and operating system type. The distribution for these resource dimensions is generated by utilizing the configuration of resources that are deployed in the various Grids including NorduGrid, AuverGrid, Grid5000, NaregiGrid, and SHARCNET [63].

Spatial Index Configuration

In this simulation, the f_{min} of logical d-dimensional spatial index is set to 4. The index space resembles a Grid-like structure, where each index cell is randomly hashed to a Grid peer based on its control point value. With dim = 4, total of 256 index cells were produced at the f_{min} level. Hence in a network that consisted of 100 GASs, on an average the responsibility of managing 2.5 index cells were assigned to each GAS.

Resource Load Indicator

The GASs/brokers encode the metric *"number of available processors"* at time τ with the resource ticket object, u_i. A coordination service utilizes this metric as the indicator for the current load on a resource, R_i. In other words, a coordination service stops sending the notifications as the number of processors available with a ticket issuer approaches *zero*.

5.4.3 Results and Observations

In our simulation, we vary the resource ticket update (RUQ) inter-arrival delay over the interval [100, 300] in steps of 100 seconds and the size of the workflow from 50 to 500 tasks. The graphs in Fig. 5.5 to Fig. 5.8 show the performance of the proposed scheduling algorithm in terms of scheduling and coordination perspective, respectively.

Scheduling perspective

As a measurement of the scheduling performance, we use the following metrics:

1. $average\ makespan$
2. $average\ coordination\ delay$
3. $average\ response\ time$
4. $average\ number\ of\ notifications$

The metric $coordination\ delay$ sums up the latencies for: (i) resource claim to reach the index cell, (ii) waiting time till a resource ticket matches with the claim, and (iii)

notification delay from coordination service to the relevant GAS. CPU time for a task is defined as the time, a task takes to actually execute on a processor.

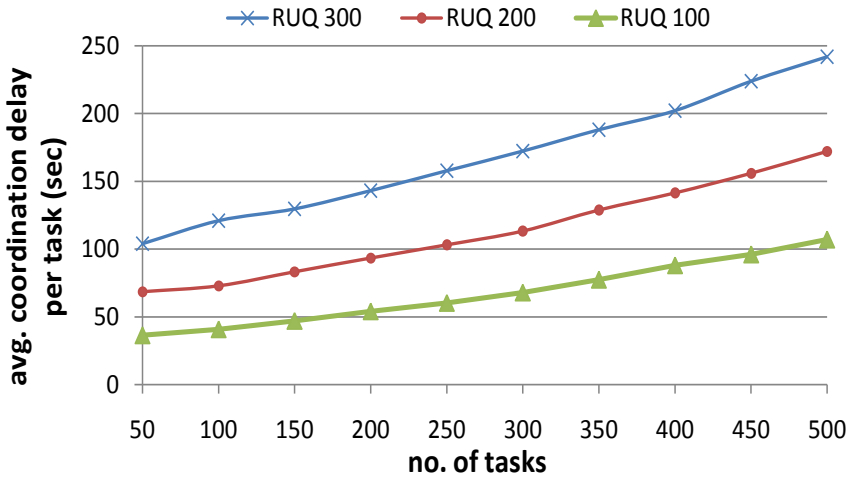
$Response\,time$ for a task is the delay between the submission time and the arrival time of execution output. Effectively, the response time includes the latencies for coordination and the CPU time. Note that, these measurements (except makespan) are collected by averaging the values obtained for each task in the system.

Definition 5.1 (Makespan): Makespan is calculated as the response time of a whole workflow, which is equal to the difference between the submission time of the entry task in the workflow and the output arrival time of the exit task in that workflow.

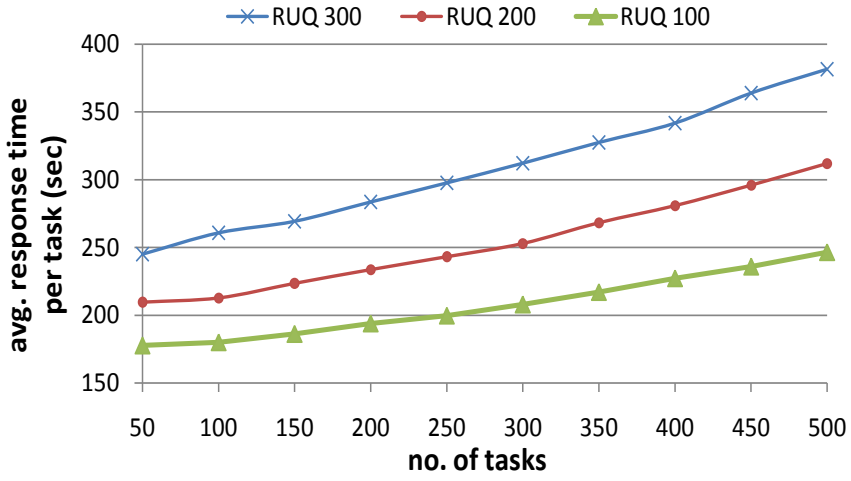
Example 5.2: Let us consider that a user at Grid site S_i wants to execute a fork-join workflow illustrated in Fig. 5.4, consisting of 11 *tasks. If GAS_i submits a claim object for task T_1 to the overlay at time $t_1 = 20$ sec and the output of task T_{11} is delivered to the user at time $t_2 = 1220$ sec, then the makespan of this workflow is $t_2 - t_1 = 1200$ sec.*

The measurement of makespan is taken by averaging over all the workflows in the system, where the completion time of a task is the difference between the time when a task has been submitted by the scheduler to a resource and the time when the output of that task has been achieved.

Fig. 5.5(a) presents the results of average coordination delay for a task with respect to the increase of the number of tasks in a workflow for different inter-arrival delays of resource information update (ticket posting frequency). The results show that at higher inter-arrival delay of tickets, the tasks in a workflow experience increased coordination delay. This happens due to the reason that in this case, the resource claim objects of the corresponding tasks have to wait for longer period of time before they are hit by ticket objects. As the task processing time (CPU time) is not affected by the ticket posting frequency, the average response time for a task shows (refer to Fig. 5.5(b)) the similar trend as coordination delay with the changes of resource information update delay. However, when the number of tasks in the workflow increases, the resource claim to ticket ratio in the system also increases. This leads to increased coordination delay for each task due to longer waiting period. Therefore, the response time of a task in the workflow is also increased, while the size of workflow increases.

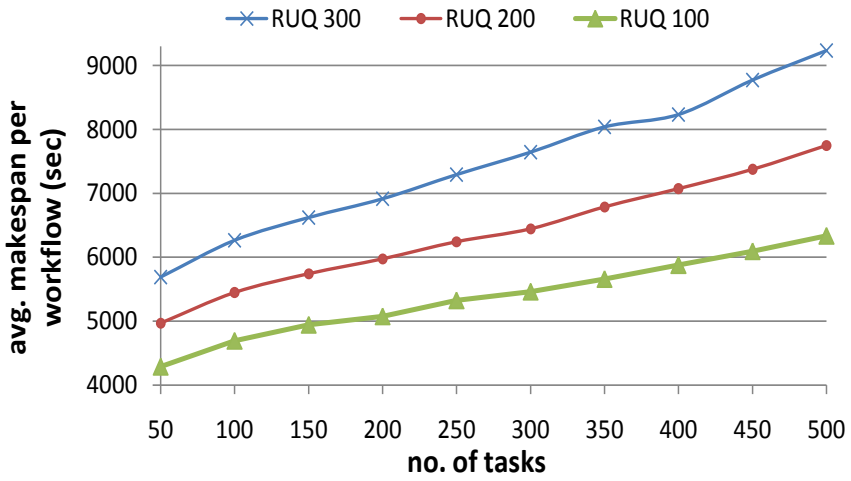


(a) Number of tasks vs. Coordination delay

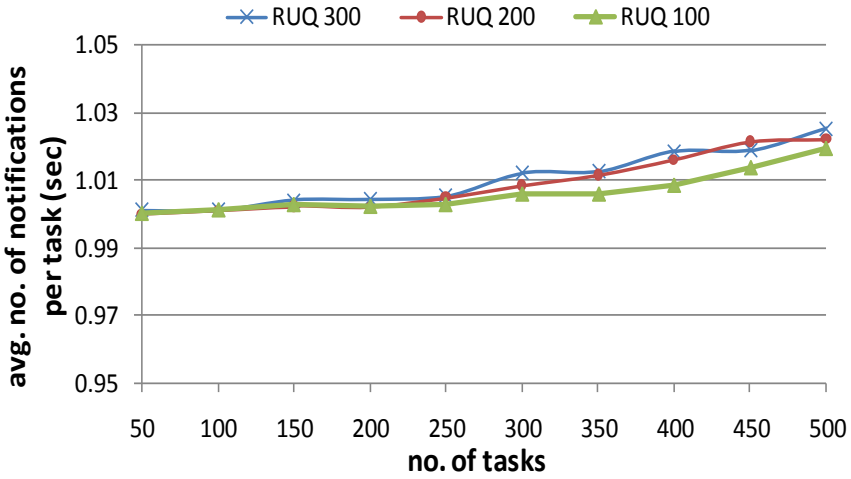


(b) Number of tasks vs. Response time

Figure 5.5: Effect of workflow size and resource information update interval on coordination delay and response time (scheduling perspective).



(a) Number of tasks vs. Makespan



(b) Number of tasks vs. Number of notifications

Figure 5.6: Effect of workflow size and resource information update interval on makespan and number of notifications (scheduling perspective).

The average makespan of the workflows also shows (see Fig. 5.6(a)) similar growth over number of tasks and ticket inter-arrival delay as reflected in coordination delay or response time. Thus, in our proposed scheduling environment, if the resources update their availability information frequently, then the workflows submitted by the users will be completed early.

The proposed scheduling approach is also highly successful in reducing the number of negotiations undertaken for the successful submission of a task (see Fig. 5.6(b)). In a centralized scheduling technique, it requires few negotiation iterations to successfully submit a task, while in this case, the average number of negotiations (notification messages) per task is about one.

Discussion: Let us consider, r number of Grid resources. Therefore, there are r number of GASs in the system. Now if each user associated with the GAS submits a workflow that consists of t number of tasks, then the total number of tasks executed on the system are,

$$T = r.t \tag{5.5}$$

Let CD_{ij} is the coordination delay of i-th task, submitted by j-th GAS in the system. Then the avg. coordination delay per task is,

$$\overline{CD} = \frac{\sum_{\substack{1 \leq i \leq r \\ 1 \leq j \leq t}} CD_{ij}}{r.t} \tag{5.6}$$

The latency for the claim object of a task, to reach the index cell (refer to Section 4.1.2) and the notification delay for that task are negligible. Thus, the coordination delay of a task mainly depends on the waiting time (in the $ClaimList$ queue) for the claim object of that task to be matched with a ticket object. In worst case, the length of the $ClaimList$ queue equals to the total number of tasks in the system. Therefore, for a fixed resource information update interval (RUQ), the waiting time of a task depends on T and we can write,

$$\overline{CD} = O(r.t)$$

$$= O(t) \tag{5.7}$$

In Fig. 5.5(a), for a particular RUQ, it is also evident that the avg. coordination delay of a task in the system is bounded by the number of task in the workflow.

The response time of a task is the summation of coordination delay and execution time of that task. Thus, the avg. response time of a task is,

$$\begin{aligned}
\overline{RT} &= \overline{CD} + \ Avg.\ CPU\ time \\
&= \overline{CD} + \frac{Avg.\ task\ size}{Avg.\ processing\ power\ of\ resources} \\
&= \overline{CD} + c_1 \ \ ; c_1 \text{ is a constant} \\
&= O(\overline{CD}) \\
&= O(t) \ \ ; \text{as } \overline{CD} = O(t)
\end{aligned} \tag{5.8}$$

Therefore, the avg. response time of a task in the system is also linearly dependant on the number of tasks in that workflow, which is shown in Fig. 5.5(b)

The average makespan of a fork-join workflow (see Fig. 5.4) consisting of t tasks and l levels, can be represented as,

$$\overline{MS} = l.\alpha.\overline{RT} \tag{5.9}$$

where the value of α depends on the number of tasks in a level (known as width, w), which can be concurrently executed depending on the availability of resources and

$$\alpha = \begin{cases} 1 & \text{in best case} \\ w/2 & \text{in average cases} \\ w & \text{in worst case} \end{cases}$$

As $\alpha = O(t), \overline{RT} = O(t)$ and in our experiments, $l = 10$, so we can write,

$$\overline{MS} = O(t) \tag{5.10}$$

Therefore, the average makespan of a fork-join workflow in the system linearly depends on the number of tasks in that workflow, which is also depicted in Fig. 5.6(a)

Moreover, for most of the tasks, the GASs/brokers in the system receive 1 notification message from the coordination service. Thus, total number of notifications for all the tasks in the system are,

$$NT = r.t + \beta \ ; \ \beta \text{ is a constant and } \beta \leq t$$

Avg. number of notifications per task,

$$\begin{aligned} \overline{NT} &= \frac{r.t}{r.t} + \frac{\beta}{r.t} \\ &= 1 + c_3 \ ; c_3 \text{ is a constant} \\ &= O(1) \end{aligned} \tag{5.11}$$

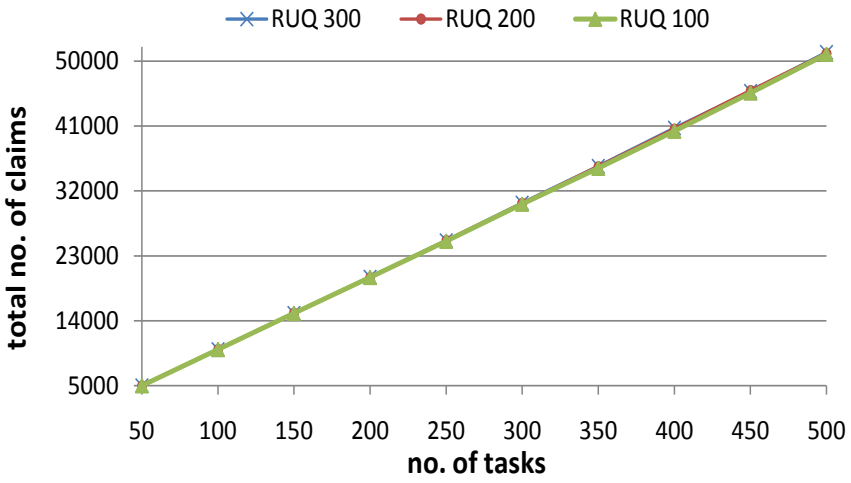
This suggests that the negotiation or notification complexity involved with this scheduling technique is $O(1)$, which we can also see in Fig. 5.6(b).

Coordination perspective

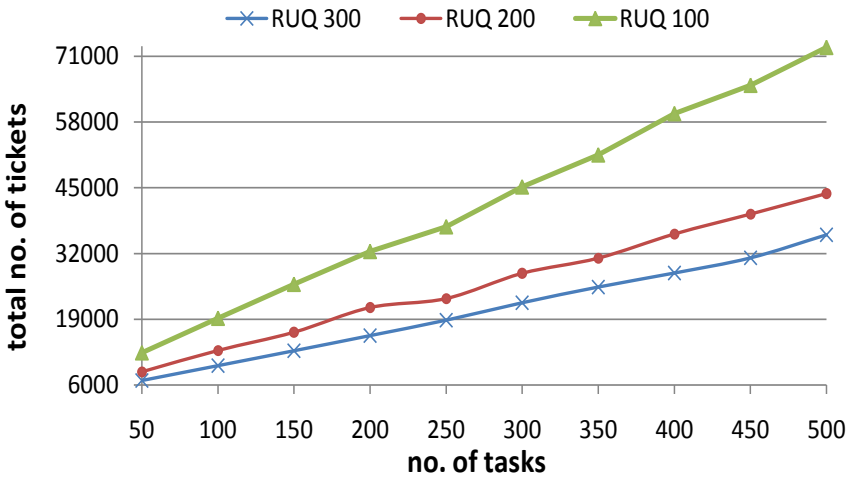
Here, we analyze the performance overhead of the DHT-based coordination space in regards to facilitating coordinated scheduling among the distributed GASs. For that, we measure the following metrics:

1. $number\ of\ routing\ hops$, undertaken per task to map claim and ticket objects to index cells;
2. $total\ number\ of\ tickets\ or\ claims$, produced in the system;
3. $total\ number\ of\ messages$, generated for the successfully mapping the coordination objects (tickets and claims) and receiving notifications.

Fig. 5.8(a) shows the number of routing hops, undertaken at different ticket injection rates for ticket and claim object across the GASs in the system for different sizes of workflow. From the figure, it is evident that the number of routing hops is not changed

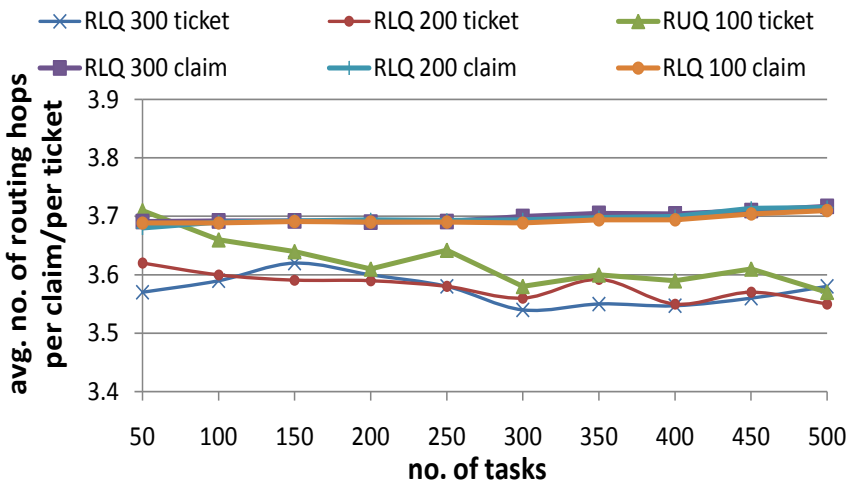


(a) Number of tasks vs. Number of claims

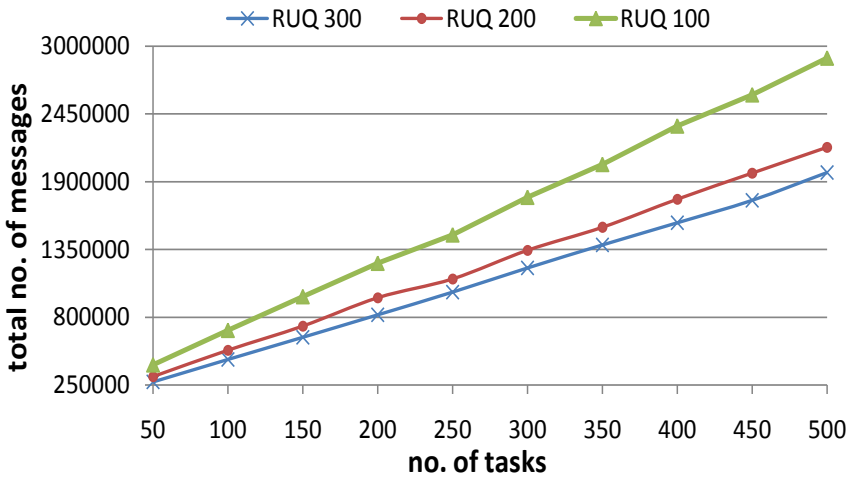


(b) Number of tasks vs. Number of tickets

Figure 5.7: Effect of workflow size and resource information update interval on number of claims and tickets (coordination perspective).



(a) Number of tasks vs. Number of hops



(b) Number of tasks vs. Number of messages

Figure 5.8: Effect of workflow size and resource information update interval on number of hops and messages (coordination perspective).

significantly with the increase of the ticket injection rate or the size of the workflow. Here, the average number of routing hops for mapping ticket or claim objects is around 3.62.

However, the number of claims, generated during the simulation, remains the same with the variation of ticket inter-arrival delay (refer to Fig. 5.7(a)). But it shows a linear growth over the increase in number of tasks in the workflow.

In Fig. 5.7(b) and 5.8(b), we show the message overhead, involved with the inter-arrival delay of ticket objects. Fig. 5.7(b) depicts the total number of ticket objects, posted by all GASs in the system with respect to increasing workflow size and ticket inter-arrival delay. In Fig. 5.8(b), we can see that as ticket inter-arrival delay and size of the workflow increase, the number of messages generated during simulation period increases. For instance, when ticket inter-arrival delay is 100 seconds and each workflow consists of 500 tasks, 72754 tickets as well as 2904113 messages are generated in the system. Thus, if the GASs publish tickets at relatively faster rate, the message overhead of the system increases substantially. But in this case, average coordination delays per task also decreases moderately (see Fig. 5.5(a)). Therefore, the ticket inter-arrival delay should be chosen in such a way that a balance between coordination delay and message overhead can exist in the system. In addition, from Fig. 5.8(b), it is evident that our system is scalable since the total number of messages is increased linearly with respect to number of tasks in the workflow.

Discussion: In our experiment, the avg. number of routing hops per claim or per ticket is,

$$\begin{aligned} HP &= 3.62 \ \text{; refer to Fig. 5.8(a)} \\ &= 6.64 - 3.02 \\ &= \log 100 - c_4 \ \text{; } c_4 \text{ is a constant} \\ \\ &= O(\log r) \ \text{; here, r = 100} \end{aligned} \tag{5.12}$$

This shows that the number of routing hops for mapping claim/ticket object is as expected in a Chord based routing space, i.e. bounded by the function $O(\log r)$.

The GASs in the system post one claim object for each task in the workflow. Thus, the total number of claim objects generated in the system is,

$$CL = r.t$$

$$= O(t) \tag{5.13}$$

Fig. 5.7(a) shows that the total number of claims generated in the system is linearly dependant on the number of tasks in the workflow.

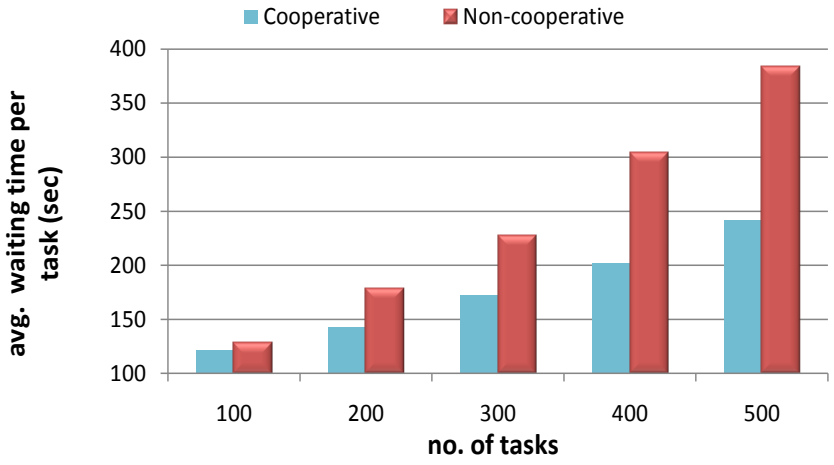
According to Ranjan et al. [102], in a federation of r heterogeneous resources, on average, a task requires HP (see Eq. 5.12) messages to be sent in the network in order to map a task to a resource. Hence given r worflows each having t tasks, the total number of messages generated to map all tasks to resources is bounded by $O((r.t)\log r)$. Further if GASs posts $\frac{1}{\lambda_t^{in}}$ tickets over a time period, then the average-case message complexity involved with mapping the ticket objects to Grid peers in the federation is bounded by the function $O(E[query\ paths]\ .\ \frac{1}{\lambda_t^{in}}\ .\ logr)$, where $E[query\ paths]$ denotes the mean number of disjoint query paths taken to map a ticket object to a Grid peer. Thus, the total number of messages generated in the system as a result of successfully scheduling all tasks of workflows in the system is,

$$O((r.t)\log r) + O(E[query\ paths]\ .\ \frac{1}{\lambda_t^{in}}\ .\ log\ r)$$

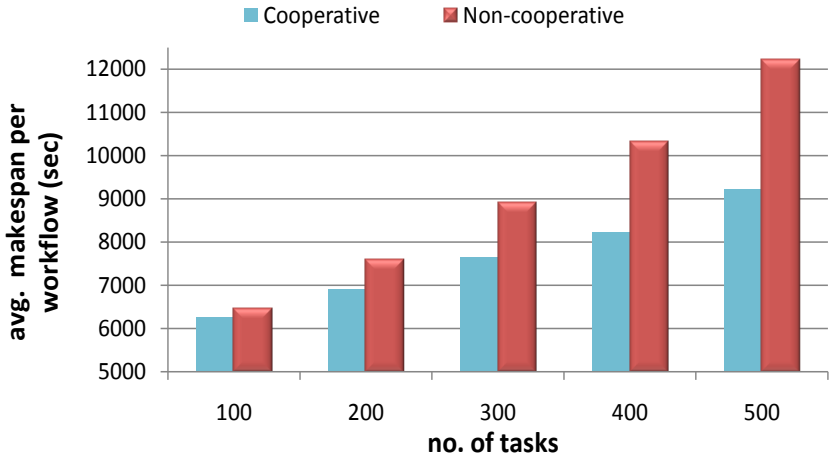
5.4.4 Comparison

Cooperative vs. Non-cooperative Load Balancing

In this section, we present the details and simulation results related to how our proposed decentralized and cooperative scheduling approach is effective in solving load-balancing problem in mapping workflow tasks to distributed Grid resources. The effectiveness of the proposed approach as regards to load-balancing is proven by conducting a comparative evaluation against a traditional non-cooperative workflow task mapping approach. In non-cooperative approach, workflow brokers operate in an independent and non-cooperative



(a) Task waiting time vs. Number of tasks



(b) Makespan vs. Number of tasks

Figure 5.9: Performance comparison of cooperative approach against non-cooperative approach with varying workflow size.

manner by submitting all the matched tasks to the first matched Grid resource, suggested by the decentralized resource discovery service in response to a resource lookup request. Further, unlike the proposed approach, the non-cooperative scheduler does not take into account the dynamic resource conditions (utilization, queue length) and priorities of other workflow brokers in the environment.

To show the performance gain achieved by applying proposed approach for mapping workflow tasks in the Grid environment, we conduct experiments by varying the size of the workflow from 100 to 500 tasks and fixing the RUQ inter-arrival delay at 300 seconds. As a measurement of the load balancing efficiency, we use the following metrics: (i) *average makespan per workflow* (see Section 5.4.3), an efficient workflow task mapping approach should be able to minimize the makespan for workflow applications across the environment; (ii) *average task waiting time per task* in the system; and (iii) *number of task executions per processor* for different Grid resources.

The metric, task waiting time is measured as the total queuing time of a single task in the system, which is equal to the difference between the submission time of a task and its execution starting time on a Grid resource. Number of task executions per processor, t_p, indicates how the task processing load is balanced across Grid resources in the environment. t_p, for a particular Grid Resource, r_i is calculated as,

$$t_p(r_i) = \frac{total\ number\ of\ tasks\ executed\ by\ r_i}{p_i}$$

Fig. 5.9(a) presents the results related to the average waiting time for a task with increase in workflow size for both the cooperative and non-cooperative scheduling approaches. The results show that if workflow brokers undertake cooperative scheduling, the tasks across the Grid environment experience relatively smaller waiting time as compared to non-cooperative scheduling. Furthermore, with the increase in workflow size (number of tasks) across the system, the performance gain achieved by applying the proposed approach is more evident. For instance, when the number of tasks in a workflow is 100, the difference between the task waiting times for cooperative and non-cooperative approaches is 7 seconds, whereas the difference increases to 142 seconds when each workflow consists of 500 tasks. Accordingly, the average makespan of the workflow also shows sim-

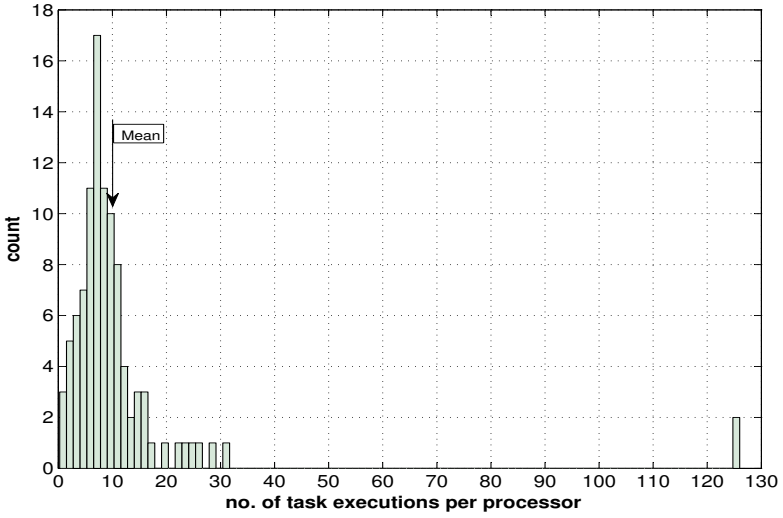
ilar improvement for cooperative scheduling approach in comparison to non-cooperative scheduling (refer to Fig. 5.9(b)). For example, when workflow size is 100, makespan is reduced by 5%, whereas makespan is decreased by 25% when the workflow consists of 500 tasks.

As the non-cooperative scheduling technique applied by the workflow brokers does not not take into account the resource behaviors (utilization, queue length) that dynamically change across the environment, resources are often overloaded and the amount of time, a task has to wait in a queue before being assigned to a processor increases. This situation is further aggravated as the workflow size scales. However, the proposed approach is able to dynamically monitor status of the resources and coordinated the distribution of load across available resources uniformly in the Grid environment, thereby minimizing task waiting time and makespan.

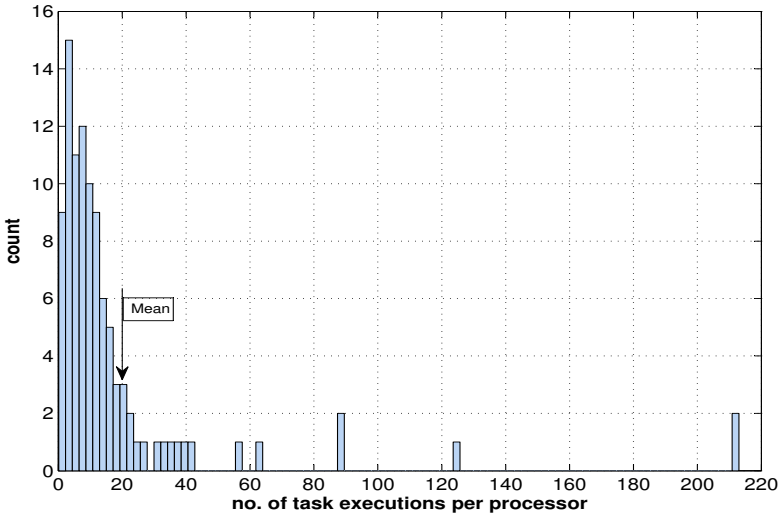
In Fig. 5.10 we show histograms of the distribution of the number of task executions per processor for different Grid resources. The distribution for non-cooperative approach shows more variability as it is more spread out and the values differ more from the center (mean value) as it is skewed. This happens because some resources are allocated more tasks to execute, while other resources are underutilized. On the contrary, the distribution for cooperative scheduling approach is bell shaped and more densely clustered around the mean. Therefore, from Fig. 5.10, it is evident that the load balancing technique employed by the cooperative scheduler is effective in terms of uniformly distributing workflow tasks accross the resources in the Grid environment. The summary statistics of these two distributions for different workflow sizes are presented in Table 5.3.

Table 5.3: Summary statistics of number of tasks (per processor) executed by Grid resources for different workflow size

Tasks	**Approach**	Min	Q_1	$Median$	Q_3	Max
100	Cooperative	0.11	0.90	2.07	3.70	26.75
100	Non-cooperative	0.09	1.06	1.80	3.53	45.75
500	Cooperative	0.25	5.65	7.97	10.78	126.00
500	Non-cooperative	0.12	4.70	9.06	15.95	213.00



(a) Cooperative scheduling (workflow size = 500)



(b) Non-cooperative scheduling (workflow size = 500)

Figure 5.10: Distribution of number of tasks (per processor) executed by Grid resources.

Decentralized vs. Centralized Coordination

In order to compare the effectiveness of our proposed decentralized and cooperative scheduling technique, we compare it against the scheduling technique with centralized coordination service. The simulation system, developed for evaluating a centrally coordinated Grid Federation for this study assumes that the coordination service is hosted on a machine, which is known to all the workflow brokers (GASs) in the system. In this setup, every workflow broker is required to submit its resource claim objects to the centralized coordination service. Similarly, all resources in the Grid Federation periodically update their usage information to the coordination service.

In case of decentralized setting (i.e. P2P spatial index), a claim object can be mapped to more than one control points based on its spatial extent or query window size. If the control points are assigned to different Grid peers in the network then the problem of duplicate or conflicting notification can arise for a given claim (i.e. a claim is hit by more than one ticket object at more than one Grid peer at the same time). This problem is quite evident in case of decentralized spatial index [26]. On the other hand, under the centralized setting, the coordination space (see Fig. 4.5) is hosted and managed by a single machine in the system and all the control points, claims and tickets are stored with it. Therefore, the problem of duplicate or conflicting notifications does not occur here and the notification complexity involved with the centralized approach is bounded by the function $O(1)$. In addition, the centralized coordination service generates schedule that can efficiently distribute the workload among all the Grid peers as it receives all the claims and tickets in the system.

The objective of the discussion of this section is to show that (i) our decentralized and cooperative scheduling technique is as efficient as the centralized coordination technique at solving the duplicate or conflicting notification problem; and (ii) the final makespan achieved per workflow as a result of decentralized coordination service is comparable to the centralized approach.

As shown in the previous section (refer to Section 5.4.3), the simulation results for our proposed approach show that the notification complexity is bounded by the function $O(1)$ (similar to the centralized approach). This is achieved due to unsubscribing the

claim object of a task from the coordination space as soon as the match notification arrives. Moreover, in Fig. 5.11, we show the average makespan, achieved per workflow in case of decentralized and centralized approaches for different sizes of workflow. From the figure, it is evident that our proposed decentralized technique is also successful in generating schedules, which are as efficient as the centralized approach. This proves that our decentralized coordination service is able to efficiently schedule the workload among the Grid peers in the system.

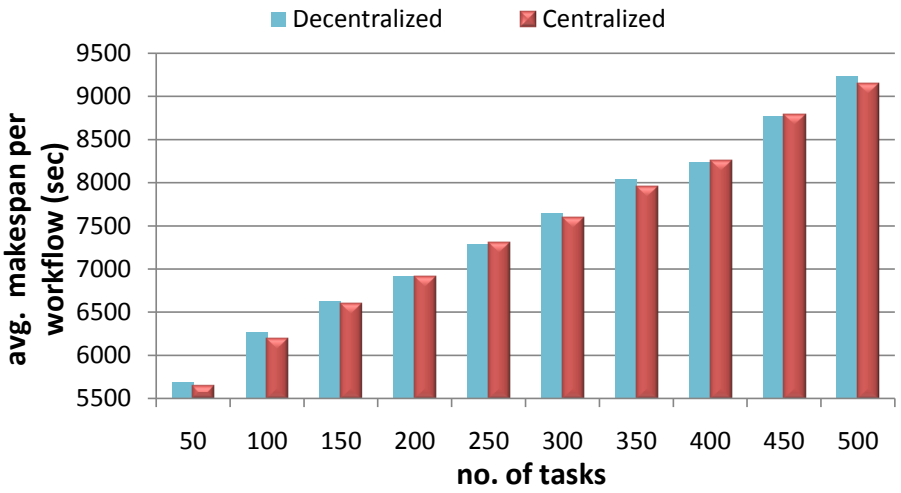


Figure 5.11: Makespan for decentralized and centralized coordination service

5.5 Related Work

The main focus of this section is to compare the novelty of the proposed work with respect to existing systems. We classify the related research in two main areas:

5.5.1 Scheduling Infrastructure

With the increasing interest in Grid workflows, many Grid workflow systems, such as Pegasus [36], Triana [118][119], Taverna [85], Condor DAGMan [76], Kepler [79], SwinDeW-G [131], Gridbus [136] and Askalon [39] have been developed in recent years. Among these systems, in terms of workflow scheduling infrastructure, SwinDeW-G and Triana utilize decentralized P2P based technique. However, the P2P communication in

SwinDeW-G and Triana is implemented by JXTA [54] protocol, which uses a broadcast technique. In this work, we use a DHT (such as Chord) based P2P system for handling resource discovery and scheduling coordination. The employment of DHT gives the system the ability to perform deterministic discovery of resources and produce controllable number of messages in comparison to using JXTA.

5.5.2 Coordination Mechanism

Jia et al. [136] proposed a workflow enactment engine, with a just-in-time scheduling system using tuple space to manage the execution of scientific workflow applications. In this system, every task has its own scheduler called Task Manager (TM), which implements a scheduling algorithm and handles the processing of tasks. The TMs are controlled by a Workflow Coordinator (WC). Besides, an event-driven mechanism with subscription-notification methods supported by the tuple space model is used to control and manage scheduling activities. In this work, although the task managers are working in a distributed fashion, they communicate with each other through the tuple space, which is implemented based on a client-server based centralized technology. Further, the WC in this system does not communicate with other WCs, managing workflow applications in the Grid. In contrast to this work, we propose a completely decentralized workflow coordinator based on a scalable P2P network model.

Yao et al. [132] have extended the aforementioned architecture with an additive Reinforcement Learning Agent (RLA) to perform the Decentralized Dynamic Workflow Scheduling using Reinforcement Learning (DDWS-RL) algorithm. DDWS-RL-enabled TMs query information from the RLA and make decision on resource selection at the time of task execution. Thus, RLA is used in the tuple space to facilitate the scheduling algorithm to be more efficient. However, this approach also involves the same architecture as the other approach, stated above, which is based on centralized client-server model with respect to resource discovery and coordination space management. In contrast, with our approach there is no central component that can prove to be a bottleneck.

Chen et al. [29] proposed a policy based coordination mechanism to manage the services provided by the peers in the Federated Service Providing (FSP) system. Peers in

FSP system share the computation resources for offering domain specific services. When a peer receives a service request that cannot be processed by itself (because either it is busy or the service type is different), it tries to find other peer in the system capable of processing the request. In order to regulate the interactions among the peers, they have proposed a recruiting protocol and a policy-driven architecture is also introduced to control the decision-making process of each peer. On the other hand, all the peers in our Grid federation system offer similar services and the coordination among the peers is managed by utilizing a DHT-based coordination service.

5.6 Conclusion

In this chapter, we have presented a decentralized and cooperative scheduling technique for workflow applications in global Grids. It leverages a DHT-based self-configuring overlay for resource coordination and a cooperative decision making strategy to achieve runtime optimization. In order to show the performance of the proposed approach against non-cooperative scheduling approach, we have conducted experiment for different sizes of workflow and the results show that our scheduling technique can reduce the makespan up to 25% by decreasing the task waiting time up to 37% and is able to balance the load among the available Grid resources significantly.

We have also compared the effectiveness of our proposed scheduling technique against the scheduling technique with centralized coordination service. According to the results obtained, our decentralized and cooperative scheduling technique is as efficient as the centralized approach at generating $O(1)$ notifications per task and the makespan achieved per workflow by our approach is also comparable to the centralized approach.

In the next chapter, we utilize the self-configuring and self-optimizing nature of this decentralized and cooperative scheduling technique to propose a reputation based dependable workflow scheduling technique in order to enable self-healing property for the workflow management system.

Chapter 6

Reputation-based Dependable Workflow Scheduling

In this chapter, a reputation-based Grid scheduling algorithm is presented to counter the effect of inherent unreliability and temporal characteristics of computing resources in large scale, decentralized Grid systems discussed in Chapter 4 and Chapter 5. The proposed algorithm can schedule workflows by dynamically adapting to changing resource conditions and offer significant performance gains as compared to traditional approaches in the event of unsuccessful job execution or resource failure. The results evaluated through an extensive trace driven simulation show that this scheduling technique can significantly reduce the makespan and successfully isolate the failure-prone resources from the system.

6.1 Introduction

In a large scale heterogeneous Grid environment, uncertainty and unreliability are facts, which are triggered by multiple factors, including: (i) software and hardware failures as the system and application scale that lead to severe performance degradation and critical information loss; (ii) dynamism (unexpected failure) that occurs due to temporal behaviours, which should be detected and resolved at runtime to cope with changing conditions; and (iii) lack of complete global knowledge that hampers efficient decision making

as regards to composition and deployment of the application elements. This chapter addresses these challenges by developing a novel self-managing [11] scheduling algorithm that aids the Grid schedulers in achieving self-healing capability by taking into account the Grid site's prior performance and behaviour for facilitating opportunistic and reliable placement of application components.

The proposed algorithm enables dependable scheduling as it is capable of dynamically monitoring and measuring reliability (i.e. availability, failure) of Grid sites and mapping workflow tasks to resources accordingly. The dependability of a Grid site is quantified using a decentralized reputation model, which computes local and global reputation scores for a Grid site based on the feedbacks provided by the scheduling services that have previously submitted their applications to that site.

The effectiveness of the proposed algorithm is appraised through a comprehensive simulation-driven analysis based on realistic and well-known application failure models to capture the transient behaviours that prevails in existing Grid-based scientific application execution environments. Further, we present a comparative evaluation that demonstrates the self-healing capability (through adaptation) of the proposed approach in comparison to Grid environments, where: (1) resource/application behaviours do not change (i.e. no failure occurs); therefore no self-management is required, and (2) transient conditions exist but runtime systems and application elements have no capability to self-adapt.

6.2 System Models

6.2.1 Grid Model

The proposed scheduling algorithm utilizes the Grid Federation [101] model discussed in Chapter 4 with regards to distributed resource organization and Grid networking. However, in order to accommodate the reputation of a Grid site for dependable scheduling, we introduce another coordination object, named *feedback*. A *feedback* is an object, sent by a GAM regarding the reputation of a Grid site in the system upon the output arrival of a previously submitted task. It contains the reputation score assigned by a Grid site for another site after a particular task execution by that site. In Table 6.1, we show an example

list of feedback objects that are sent to the coordination service by individual Grid service consumers.

Table 6.1: Feedbacks sent by different Grid sites to the coordination service

Feedback ID	From	For	User ID	Workflow ID	Task ID	Score
002	S_3	S_1	1	1	4	1.0
040	S_2	S_9	1	2	6	0.5
100	S_2	S_9	1	2	9	0.42
251	S_5	S_{10}	1	3	89	0.5

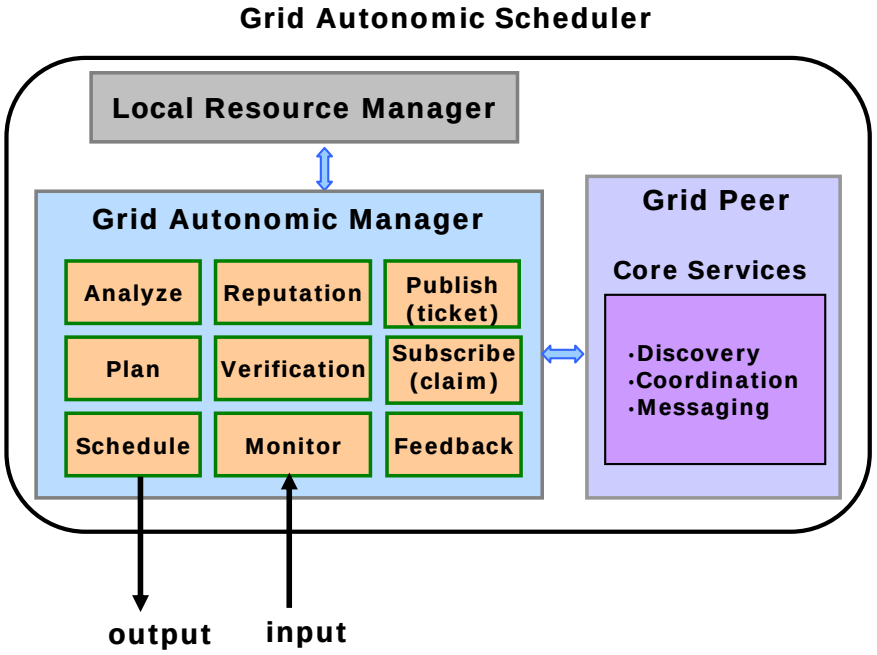


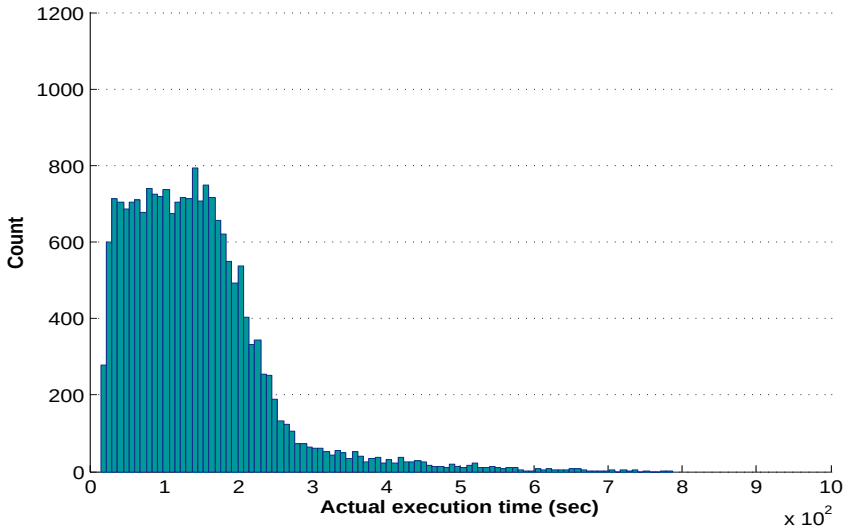
Figure 6.1: The architecture and components of Grid Autonomic Scheduler

6.2.2 Application Model

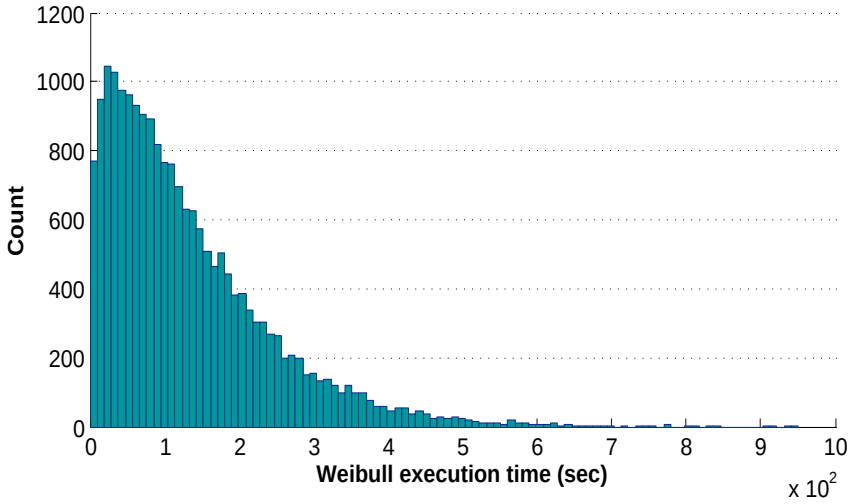
We consider the Scientific workflow applications as the case study for the proposed scheduling approach. The modeling of these kind of applications is described in Section 4.1.3 of Chapter 4.

Table 6.2: Notations: Grid, reputation, failure models, and metrics

Symbol	Meaning
Grid	
n	number of sites or GASs in the Grid system
S_i	i-th Grid site in the system
GAS_i	i-th GAS in the system
Reputation	
$succ(i,j,k)$	output of result verification function for task T_k of S_j executed by S_i.
$feed(i,j,k)^t$	feedback score of task T_k from S_j for S_i after t transactions.
NF_i	total number of negetive feedbacks given by other sites for S_i.
TF_i^t	transaction feedback value for S_i after t transactions by S_i.
$TF_{i,j}^t$	transaction feedback value from S_j for S_i after t transactions.
GR_i^t	global reputation of S_i after t transactions.
$LR_{i,j}^t$	local reputation of S_i according to S_j after t transactions.
M_{LR}	local reputation matrix.
M_{GR}	global reputation matrix.
$LR_{initial}$	initial local reputation value of each site.
$GR_{initial}$	initial global reputation value of each site.
R_{th}	reputation threshold of a site for a task to be mapped by scheduler.
$\tau_{refresh}$	time interval after which initial value is assigned to reputation score of a site.
Failure	
fp_i	task failing probability of Grid site S_i.
X_Y	failure distribution, where $X\%$ sites fail task with probability between Y and $Y + 0.1$.
Metrics	
$M_{i,j,k}$	makespan of k-th workflow submitted by j-th user of i-th Grid site.
$M_{average}$	average makespan per workflow in the system.
F_i	number of tasks failed by site S_i.
F_{total}	total number of tasks failed in the system.
SCH_i	number of tasks scheduled by GAS_i.
SCH_{total}	total number of tasks scheduled in the system.
$\rho_{M_{average},F_{total}}$	Pearson's correlation coefficient between $M_{average}$ and F_{total}.



(a) Histogram of task execution time based on experiments (Average execution time = 141 sec).



(b) Histogram of task execution time based on weibull distribution ($\beta = 1.2$, $\eta = 141$).

Figure 6.2: Distribution of task execution time.

6.2.3 Failure Model

The Weibull distribution [66] is one of the most commonly used distributions in reliability engineering and has become a standard in reliability textbook for modeling time-dependant failure data. Therefore, in this work, we use a 2-parameter weibull distribution to determine whether a task execution is subject to failure or success in the system. The 2-parameter weibull distribution is generally characterized by two parameters: shape parameter β and scale parameter η. Fig. 6.2 shows the actual and weibull distribution of the task execution time in the system.

After a task has finished its execution on a resource, the execution time or the computational cost of that task is measured. If it falls within a certain range of the weibull distribution, then the task is considered as likely-to-fail. A task execution may fail for various reasons (e.g. the resource does not have appropriate libraries installed, executables are outdated or the resource has been restarted before sending all the output files). Thus, whether a task is likely to be failed is derived from weibull distribution and the logic for determining this is illustrated in Fig. 6.3. Next, if a task is likely-to-fail, then whether it will be considered as a failed task further depends on the failure probability fp_i of the resource at site S_i that has been assigned to execute the task. For this, we generate a uniform random number between 0 and 1. If the value of this random number is less than fp_i, then the task is failed, otherwise it is successful.

Definition 6.1 (Failure Probability): Failure Probability, fp_i is defined as the likelihood or chance that the resource at Grid site S_i will fail the execution of a workflow task that is likely-to-fail in the system.

Example 6.1: Let us consider that failure probability of the resource at Grid site S_2 is 0.57. Hence, S_2 will fail 57 of the 100 likely-to-fail tasks assigned to it for execution.

6.3 Proposed Methodology

6.3.1 Distributed Reputation Management

In this section, we propose the key methods related to the distributed reputation management and its application to dependable scheduling.

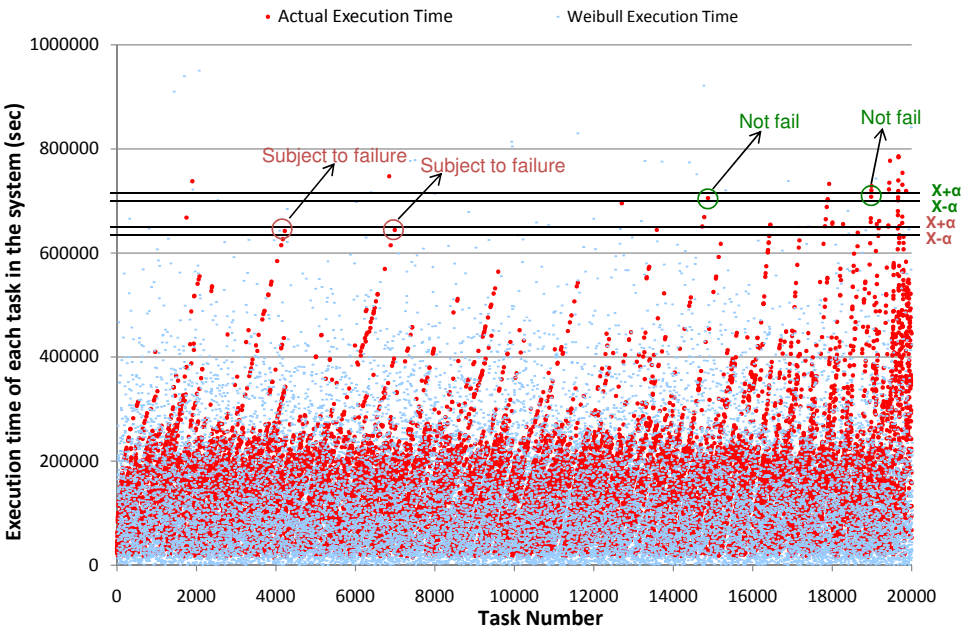


Figure 6.3: Determination of tasks likely-to-fail based on the distribution of experimental and Weibull task execution time.

In a fully decentralized and distributed Grid overlay, the P2P reputation system calculates the reputation score for a Grid site S_i by considering the opinions (i.e. feedbacks) [65, 140] from all the Grid sites $\in \{S_1, S_2, \ldots, S_n\}$, who have previously interacted with S_i. After a Grid site S_j completes a transaction with another Grid site S_i, S_j provides its feedback for S_i to the overlay, which is utilized to compute the reputation of S_i. This reputation value drives future application scheduling decision making in choosing S_i for task execution. A Grid site, which accumulates higher reputation in the system is expected to be popular in the overlay. Over the period of time, the distributed scheduling services (GASs) in the system are more likely to prefer that site in the future for placement of tasks. On the other hand, a Grid site that performs badly over a period of time would accumulate comparatively lower reputation and will eventually be shunted out of the system, i.e. would receive none or very few job submissions from the schedulers (GAS).

In the proposed approach, the overlay maintains two reputation scores for each Grid site: (i) Global Reputation (GR) and (ii) Local Reputation (LR). Here, the GAS service (on behalf of local Grid site and users) rates the Grid sites, to which it submits a task,

after every successful transaction (task completion) or unsuccessful transaction (task failure) based on a feedback function, $feed(i, j, k)$. The local and global reputation scores for Grid sites are stored within the distributed overlay in the form of local and global reputation matrix. These values are recursively aggregated from the feedback scores after each transaction and utilized by the scheduling algorithm to dynamically quantify the reliability of the sites.

Feedback Generation

GAS services can use a variety of rating functions based on system consensus for computing the feedback value. Some of the example functions can include the model used by *eBay* system. The reputation scheme in eBay is simple: $+1$ for a good or successful transaction, -1 for a poor or failed feedback, and 0 for a neutral or don't-care feedback. In this model, the feedback score has three discrete values, which evaluate the result of a transaction. However, this model does not incorporate different types of behaviour of the participating entities (e.g. an entity is failing transactions of only a particular entity, an entity is failing transactions only at the beginning or an entity is generating successful and unsuccessful transactions alternately) into the feedback score, which is required to be considered in case of heterogeneous and dynamic resource sharing Grid environments.

In our feedback model, the GAS service at site S_j computes the feedback, $feed(i, j, k)$ for a Grid site S_i dynamically after each transaction (i.e. S_i completes execution of a task T_k submitted by S_j), and we assume that the generated feedbacks are always correct. First, S_j verifies the output of a task returned by S_i using the result verification function $success(i, j, k)$ that assigns a value $\in \{0,1\}$, where 0 represents an unsuccessful/failed task execution and 1 represents a successful task execution. A task execution may fail for various reasons (e.g. the resource does not have appropriate libraries installed, executables are outdated or resource has been restarted before sending all the output files). The result verification function is represented as,

$$success(i, j, k) = \begin{cases} 1 & \textit{if task execution is sucessful} \\ 0 & \textit{if task execution is failed} \end{cases} \tag{6.1}$$

Then S_j generates the feedback score based on the value assigned by result verification

function. If the assigned value is 1, feedback score is 1; on the other hand, if the assigned valued is 0 then the feedback score is calculated from an exponential distribution. The output given by the exponential function (refer to Fig. 6.4) is varied over the number of failed transactions between the corresponding two Grid sites. The objective of using this exponential function is to give a Grid site greater opportunity to execute tasks at the beginning so that it is not shunted out of the system after only few failed transactions. However, if a site continues to fail more transactions, the value for exponential function approaches 0. Thus, if $F_{i,j}$ is the number of unsuccessful task executions by S_i with S_j, the feedback score for task T_k, after t transactions by S_i with S_j can be represented as,

$$feed(i,j,k)^t = \begin{cases} 1 & \text{if success(i,j,k) = 1} \\ \alpha_f^{\frac{1}{\beta_f} F_{i,j}} & \text{if success(i,j,k) = 0} \end{cases} \tag{6.2}$$

where $0 < \alpha_f \leq 0.5$ and $\beta_f \in \{1, 2, 3\}$.

If the feedback score given by a Grid site S_j is 1, we consider it as Positive Feedback (PF), whereas a Negative Feedback (NF) is attained if feedback score is less than 1.

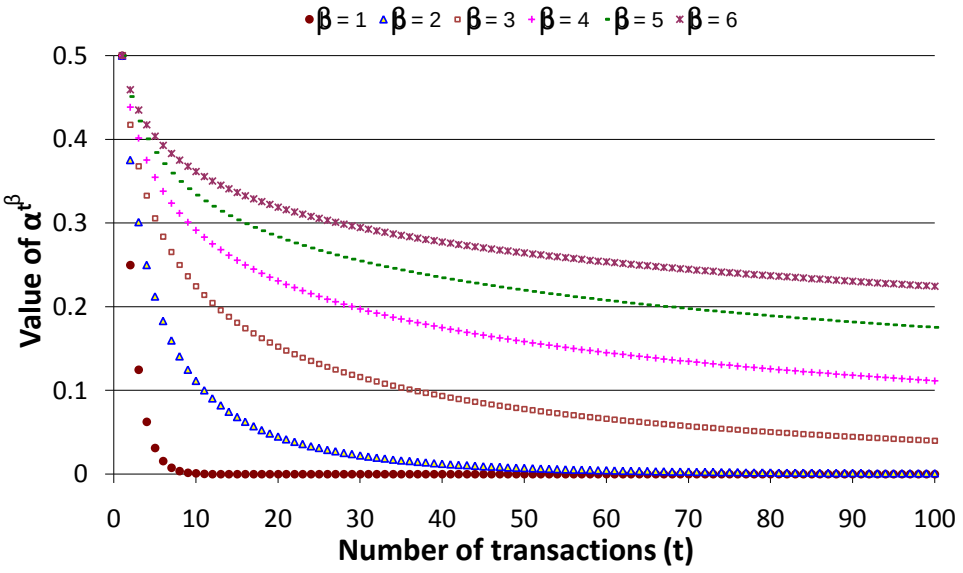


Figure 6.4: The growth of α^{t^β} over the number of transactions, t for different values of β. Here, $\alpha = 0.5$.

Global Reputation Calculation

The Global Reputation (GR) of a site is a statistical reputation that is calculated by averaging all the feedbacks given by the GAS services of other Grid sites for their tasks executed at that site. Once the overlay receives a feedback, it computes the Transaction Feedback (TF) for that feedback. The value of TF depends on whether the feedback is positive or negative. If negetive feedback is received, TF is same as the feedback value. However, if feedback is positive, the value of TF is computed from an exponential distribution (refer to Fig. 6.4), where the output value is varied over the total number of negative feedbacks received by the corresponding Grid site. The purpose of using this distribution is to allow a Grid site to accrue a higher value of GR only if it executes more successful tasks than failed tasks. So, if it fails very few transactions, the output of the exponential function reaches 1 accordingly. Thus, if NF_i is the total number of negative feedbacks given by other sites for S_i, the transaction feedback value after t transactions by S_i can be calculated as,

$$TF_i^t = \begin{cases} feed(i,j,k)^t & \textit{if (negative feedback) or (positive feedback and } NF_i \textit{ is 0)} \\ \{(1-\alpha_p)+\alpha_p^{NF_i^{\frac{1}{\beta_p}}}\} \times feed(i,j,k)^t & \textit{if (positive feedback) and } (NF_i > 0) \end{cases} \tag{6.3}$$

where $0.5 \leq \alpha_p < 1.0$ and $\beta_p \in \{4, 5, 6\}$.

The GR of a particular Grid site is calculated by taking the average of aggregated TFs form other sites. Initially GR is assigned a value $GR_{initial}$ that is greater than or equal to the reputation threshold R_{th}. Afterwards, it is dynamically changed based on the TF computed after every transaction. Thus, GR of a Grid site, S_i after total t number of transactions with other sites is represented as,

$$GR_i^t = \begin{cases} GR_{initial} & \textit{if } t = 0 \\ \frac{GR_i^{t-1} \times t + TF_i^t}{(t+1)} & \textit{if } t > 0 \end{cases} \tag{6.4}$$

The GR value of each Grid site is stored in a matrix. At any instance of time, the DHT-based distributed overlay maintains $n \times 1$ global reputation matrix M_{GR} (refer to Fig. 6.5(a)) for all the Grid sites $S_i \in \{1,2,\ldots,n\}$ that is updated dynamically after every transaction in the system. This M_{GR} is utilized by the distributed scheduler for mapping tasks to the Grid sites based on their reputation values.

$$M_{GR} = \begin{pmatrix} 0.85 \\ 0.90 \\ 0.70 \end{pmatrix} \begin{matrix} S_1 \\ S_2 \\ S_3 \end{matrix} \qquad M_{LR} = \begin{matrix} S_2 & S_2 & S_2 \end{matrix} \begin{pmatrix} 0.95 & 0.82 & 0.75 \\ 0.75 & 0.98 & 0.82 \\ 0.88 & 0.56 & 0.76 \end{pmatrix} \begin{matrix} S_1 \\ S_2 \\ S_3 \end{matrix}$$

(a) Global reputation matrix (b) Local reputation matrix

Figure 6.5: Reputation matrix for three Grid sites (S_1, S_2, S_3).

Local Reputation Calculation

Sometime, considering only GR of a Grid site for mapping tasks, can not guarantee dependable scheduling. For example, the resource at a site S_i may fail tasks submitted by only a particular Grid site S_j. In this case, as S_j successfully executes tasks submitted by other Grid sites, its GR is high. So, the scheduler may still map the tasks submitted by S_j to S_i. Therefore, we introduce another reputation score, Local Reputation (LR) for a Grid site.

Similar to GR, LR is calculated as an average of the feedback values except it considers feedbcks from only one Grid site. TF for computing LR also follows the same function as generating TF for GR. Therefore, if $NF_{i,j}$ is the number of negative feedbacks given by S_j for S_i after t transactions with S_i, the transaction feedback value can be calculated as,

$$TF^t_{i,j} = \begin{cases} feed(i,j,k)^t & \textit{if (negative feedback) or (positive feedback and } NF_i \textit{ is 0)} \\ \{(1-\alpha_p)+\alpha_p^{NF_{i,j}^{\frac{1}{\beta_p}}}\} \times feed(i,j,k)^t & \textit{if (positive feedback) and } (NF_i > 0) \end{cases} \tag{6.5}$$

where $0.5 \leq \alpha_p < 1.0$ and $\beta_p \in \{4, 5, 6\}$.

Now, the LR of a Grid site, S_i according to S_j, after t number of transactions with S_j is represented as,

$$LR^t_{i,j} = \begin{cases} LR_{initial} & \textit{if } t = 0 \\ \frac{LR^{t-1}_{i,j} \times t + TF^t_{i,j}}{(t+1)} & \textit{if } t > 0 \end{cases} \tag{6.6}$$

The LR values of each Grid site in regards to other sites are kept in a $n \times n$ local

reputation matrix M_{LR} (refer to Fig. 6.5(b)), which is stored in the overlay and updated dynamically after every transaction between the corresponding sites. Similar to M_{GR}, M_{LR} is also utilized by the distributed scheduler for mapping tasks to the Grid sites based on their reputation values.

An example scenario of local and global reputation calculation in the distributed coordination space is depicted in Fig. 6.6.

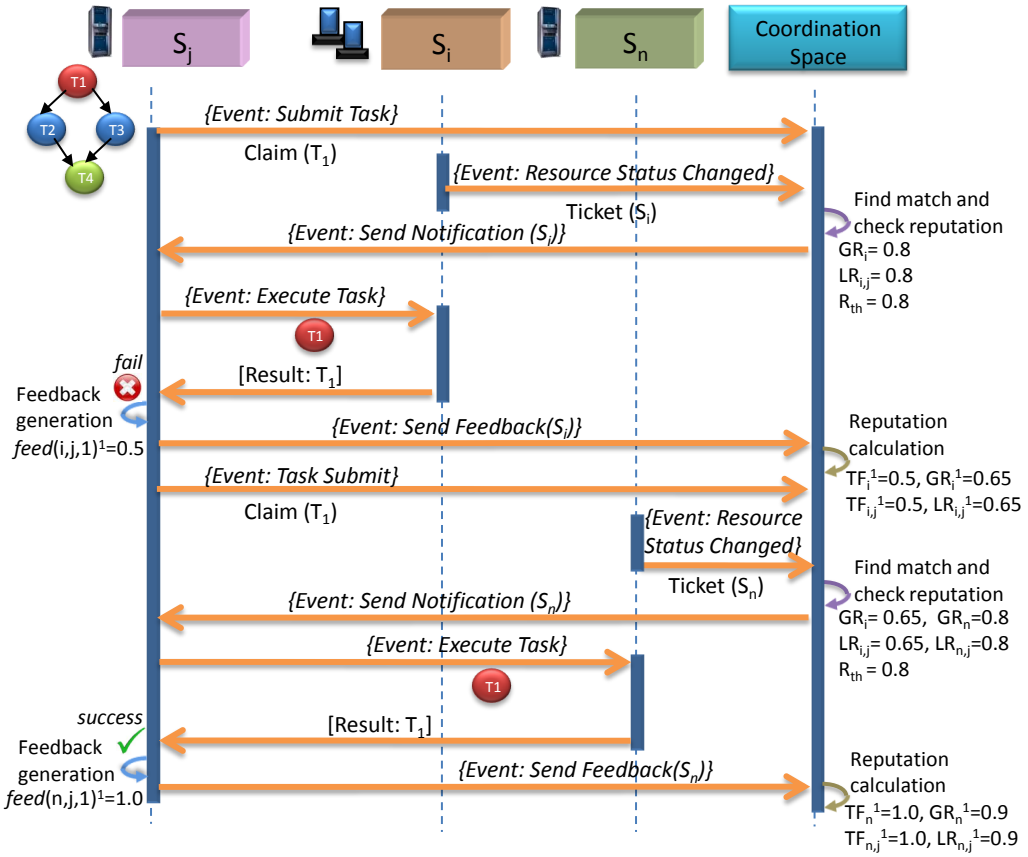


Figure 6.6: Interaction among different Grid entities in reputation based dependable workflow scheduling approach.

6.3.2 Distributed Workflow Management

In this section, we provide the description of the algorithms that have been devised for task scheduling and resource provisioning in order to achieve reputation-based workflow management.

Algorithm 5 TASK SCHEDULING AT GAS

1: **PROCEDURE**: Event- User Workflow Submit
2: Input: Workflow W
3: **begin**
4: Calculate rank value for each task using EFT heuristic
5: Generate *Ready TaskList* for W
6: Submit *Ready* tasks for execution
7: **end**
8: **PROCEDURE**: Event- Task Finish Notification
9: Input: Task T_k, Workflow W
10: **begin**
11: Update dependency list of each task in $TaskList$
12: Generate *Ready TaskList* for W
13: Submit *Ready* tasks for execution
14: **end**
15: **PROCEDURE**: Generate Feedback
16: Input: Task T_k, Site S_i
17: **begin**
18: Verify output of T_k by (6.1)
19: **if** $success(i, j, k) = 0$ **then**
20: $F_{i,j} \leftarrow F_{i,j} + 1$
21: **end if**
22: Calculate feedback score for T_k by (6.2)
23: **end**

Task Scheduling

Here, we discuss about the task scheduling algorithm (refer to Algorithm 5) that is undertaken by a GAS in Grids on arrival of a job or workflow. When a user submits a workflow application W, the GAS calculates the priority of each task (line 4). Earliest Finish Time (EFT) [122] heuristic is used to calculate task priorities by traversing the task graph in Breadth First Search (BFS) manner. Once the rank values are calculated, the GAS generates *Ready* tasks in the *TaskList* based on the dependency of each task and put them into the *Ready TaskList* (line 5). Finally, GAS submits the *Ready* tasks for execution (line 6).

Further, when the GAS receives a notification message from site S_i stating task T_k has finished execution, it first updates the dependency lists of the tasks that are dependant on T_k (line 11); then it computes the *Ready* tasks at that moment and submits those for execution (line 12-13). Next, it generates feedback for the transaction with S_i (line 15). In order to do that it first verifies the output of T_k using the result verification function

represented by Eq. (6.1) (line 18). If output of the function is 0, $F_{i,j}$ is incremented by one (line 20). Then it calculates the feedback score for this transaction by Eq. (6.2) (line 22).

Resource Provisioning

Algorithm 6 RESOURCE PROVISIONING AT COORDINATION SPACE

1: **PROCEDURE**: Event- Claim Submit
2: Input: Claim r_k
3: **begin**
4: $ClaimList \leftarrow ClaimList \cup r_k$
5: **end**
6: **PROCEDURE**: Event- Ticket Submit
7: Input: Ticket u_i from Resource R_i
8: **begin**
9: **if** $GR_i \geq R_{th}$ **then**
10: $ClaimList_m \leftarrow$ list of claims in $ClaimList$ that are matched with u_i
11: Sort $ClaimList_m$ in descending order of task's rank
12: $index \leftarrow 0$
13: **while** R_i is not over-provisioned **do**
14: **if** $LR_{i,j} \geq R_{th}$ **then**
15: Send notification of match event to resource claimer $ClaimList_m$[index]
16: Remove $ClaimList_m$[index]
17: $index \leftarrow index + 1$
18: **end if**
19: **end**
20: **end if**
21: **end**
22: **PROCEDURE**: Event- Feedback submit
23: Input: Feedback from site S_j
24: **begin**
25: **if** feedback is negetive **then**
26: $NF_i \leftarrow NF_i + 1$
27: $NF_{i,j} \leftarrow NF_{i,j} + 1$
28: **end if**
29: Calculate TF_i and $TF_{i,j}$ by (6.3) and (6.5)
30: Calculate GR_i and $LR_{i,j}$ by (6.4) and (6.6)
31: Update M_{GR} and M_{LR}
32: **end**

The details of the decentralized resource provisioning algorithm (refer to Algorithm 6) that is undertaken by the P2P coordination space is presented here. When a resource

claim object r_k arrives at the coordination service, it is added to the existing claim list, *ClaimList* by the coordination service (line 1-5). When a resource ticket object u_i arrives at coordination service, the list of resource claims ($ClaimList_m$) that overlap or match with the submitted resource ticket object is computed (line 6-10) if global reputation of that resource is greater than or equal to the reputation threshold R_{th}. The overlap signifies that the task associated with the given claim object can be executed on the ticket issuer's resource subject to its availability.

Then the coordination service sorts the claim objects in $ClaimList_m$ in descending order according to their rank value (line 11). From the $ClaimList_m$, the resource claimers are selected one by one based on their rank value (higher rank first) and notified about the resource ticket match if local reputation of ticket issuer against the resource claimer is greater than or equal to R_{th} and until the ticket issuer is not over-provisioned (line 13-19).

When a feedback object is arrived at coordination service, first it is decided whether the feedback is negetive or positive. If feedback is negetive, NF_i and $NF_{i,j}$ are incremented by one (line 25-27). Then local and global reputation scores are calculated consequently (line 29-30). Finally, the local and global reputation matrices that are stored in coordination space are updated by the coordination service (line 31).

Time Complexity

This section analyses the computational tractability of the approach by deriving several time complexity bounds to measure the computational quality. Using the example for Grid workflow application model, we analyse the complexity of calculating task rank, feedback generation, and reputation scores (see Algorithm 5). These complexities are further aggregated to model a composite function that represents the overall complexity.

We consider a Grid infrastructure consisting of n number of Grid sites. Every Grid site S_i instantiates a service GAS_i. This implies that there are total n number of GAS services in the infrastructure that are continuously injecting task to resource mapping requests in form of Claim Objects (refer to line 1 in Algorithm 6). We assume every user submits a workflow application consisting of T number of tasks and E number of dependencies among the tasks to its local GAS service. Then the complexity of calculating rank values of all the tasks using EFT heuristic through BFS is $O(E+T)$ [33]. Further, if an adjacency

list is used to handle the dependencies, then the complexity of generating *Ready* tasks and updating dependency list is $O(E)$ [33].

Next, we derive the time complexity of generating feedback in Algorithm 5 (lines 15-23). After a GAS service receives the output of a submitted task, it has to compute a feedback score, which is required to be reported to coordination service. The feedback score calculation involves few mathematical computation (see Eq. 6.2); thus, it involves constant complexity of $O(1)$. Therefore, the overall time complexity of Algorithm 5 is $O(E+T)$.

In worst case, $ClaimList$ in the Algorithm 6 can contain $n.T$ number of enteries. So the complexity of sorting the $ClaimList$ is $O((n.T)\log(n.T))$ (through the implementation of merge sort algorithm) and finding out the total number of matches is $O(n.T)$. Calculating the number of resource claimers that has to be notified about the matches also requires $O(n.T)$ steps in worst case.

Every new feedback score submitted by GAS services needs to be aggregated into global reputation score (using Eq. 6.4). Similar to feedback computation, updating global reputation score also involves series of mathematical steps. Hence, the overall complexity of computing or updating global reputation score is constant, $O(1)$.

Finally, the adjacency matrix also handles the reputation matrices. Here, updating M_{GR} and M_{LR} is bounded by $O(1)$. Thus, the overall complexity of Resource Provisioning algorithm is $O((n.T)\log(n.T))$.

6.3.3 Scheduling Example

This section provides an example scenario of the process of task scheduling and distributed reputation management. The key steps involved with the proposed scheduling approach (see Fig. 6.7) are as follows:

1. A user submits his task to the local GAS service at site S_u.

2. Following this, the GAS inserts a claim object to the DHT-based overlay to locate a *dependable* and *available* Grid site (resource) that has reasonable reputation rating (above reputation threshold) in the system.

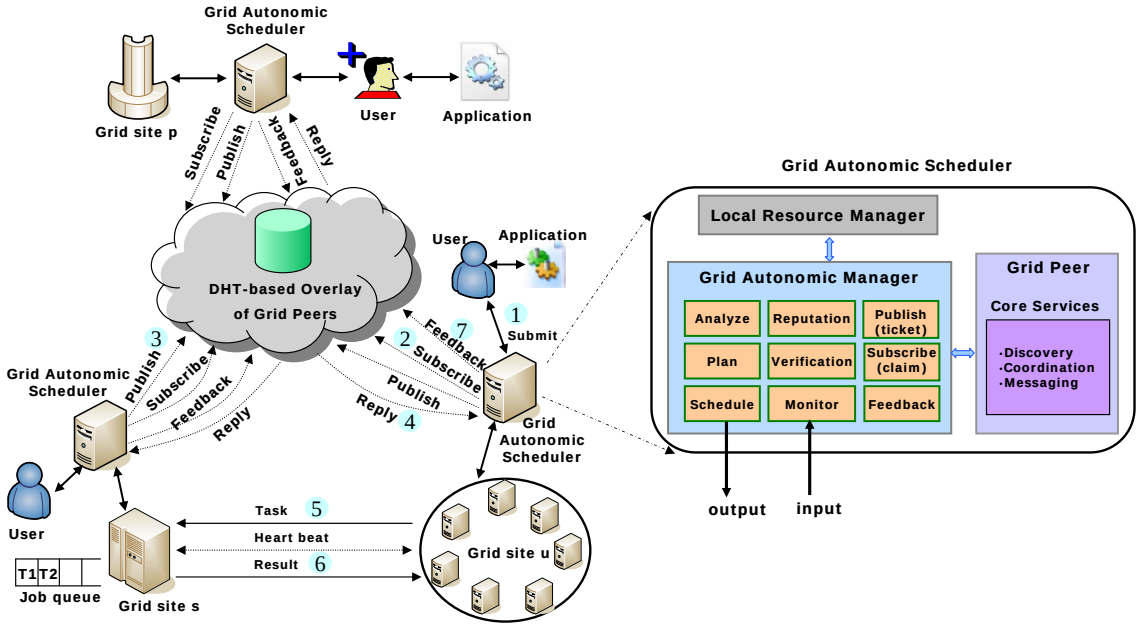


Figure 6.7: Reputation-based dependable scheduling example. Grid sites p, l, s, and u are managed by their respective Grid Autonomic Scheduler services.

3. The GAS, GAS_s at site S_s submits a ticket object to the overlay encapsulating the information about status (availability) of the local resource.

4. The overlay undertakes the decentralized matchmaking mechanism and discovers that the resource ticket issued by Grid site S_s matches with the resource description and reputation rating currently specified by claim object inserted by site S_u. Thus, a match notification message is sent to S_u.

5. Next, GAS_u sends the task to site S_s. While the application is being processed, GAS_u periodically monitors the execution progress by sending *IsAlive* messages to S_s. IsAlive messages allow the GAS services to detect the hardware and network link failure related to the site S_s.

6. Once the execution of the task is finished, S_s returns the output to GAS_u.

7. Finally, GAS_u performs the result verification for the received output, computes the feedback score for S_s and reports to the overlay. The feedback score is aggregated to the local and global reputation scores for S_s using the proposed decentralized and distributed reputation model, described in Section 6.3.1.

6.4 Performance Evaluation

6.4.1 Simulation Setup

Similar to Chapter 5, our simulation infrastructure is created by combining two discrete event simulators namely *GridSim* [22], and *PlanetSim* [49].

Workload Configuration

In this study, we also consider fork-join workflow (see Fig. 5.4) and an example of such workflow is WIEN2K [18], which is a quantum chemistry application developed at Vienna University of Technology. We vary the number of tasks in a workflow from 100 to 500 during the experiments and the size of each task is randomly generated from a uniform distribution between 50000 MI (Million Instructions) to 500000 MI. Further, we assume that workflows are computation intensive. Thus, the data dependency among the tasks in the workflow is negligible. In the Grid federation, each site has one user and each submits one workflow for execution.

Network Configuration

The experiments run a Chord overlay with 32 bit configuration (number of bits utilized to generate node and key ids). The total number of GAS/broker in the system is 64. Further, network queue message processing rate is fixed at 4000 messages per second and message queue size is fixed at 10^4.

Resource Claim and Ticket Injection Rate

In this work, the injection rate for the resource tickets is every 200 seconds [104]. Spatial extent of both resource claims and ticket objects lie in a 4-dimensional attribute space. These attribute dimensions include the number of processors, their speed, architecture and operating system type. The distribution for these resource dimensions is generated by utilizing the configuration of resources that are deployed in various Grids including NorduGrid, AuverGrid, Grid5000, NaregiGrid, and SHARCNET [63].

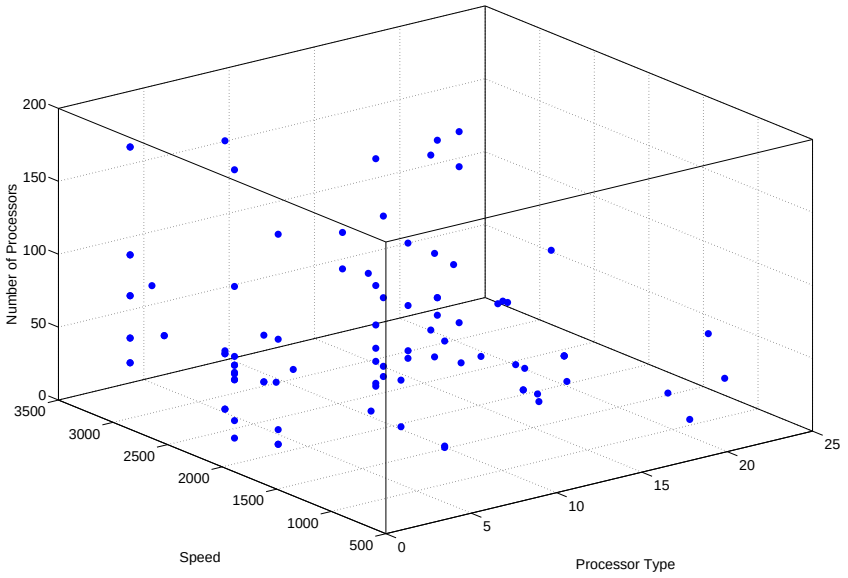


Figure 6.8: Resource configuration and Ticket data distribution.

Reputation Configuration

The values of the parameters for configuring reputation based scheduling in our experiment are listed in Table 6.3.

Table 6.3: Reputation parameters

Parameter	Value	Parameter	Value
α_f	0.5	$LR_{initial}$	0.8
β_f	2.0	$GR_{initial}$	0.8
α_p	0.5	R_{th}	0.8
β_p	5.0	$\tau_{refresh}$	1000 sec

Failure Configuration

In our experiments, the values of weibull shape and scale parameters β and η are 1.2 and 141 respectively, where the mean execution time of a task in the system is equal to 141 sec.

Along with this Weibull distribution, we also generate a set of resource failure distributions, X_Y by incorporating resource failure probability fp, where X represents the percentage of resources likely to fail tasks in the system and Y represents the probability

of failure. For instance, if X is 20 and Y is 0.4, then 20% of resources in the system may fail tasks with the probability (fp) between 0.4 and 0.5. The resource failure distributions, we use in the experiment are as follows:

$\boldsymbol{X}$_$\mathbf{0.1}$: $\mathbf{0.1 \leq fp < 0.2}$; $\boldsymbol{X}$_$\mathbf{0.3}$: $\mathbf{0.3 \leq fp < 0.4}$

$\boldsymbol{X}$_$\mathbf{0.5}$: $\mathbf{0.5 \leq fp < 0.6}$; $\boldsymbol{X}$_$\mathbf{0.7}$: $\mathbf{0.7 \leq fp < 0.8}$

$\boldsymbol{X}$_$\mathbf{0.9}$: $\mathbf{0.9 \leq fp < 1.0}$

Some example failure distributions are presented in Table 6.4 and Table 6.5.

Table 6.4: Example failure distributions (25_Y)

ResourceID	25_0.1	25_0.3	25_0.5	25_0.7	25_0.9
1	0	0	0	0	0
2	0	0	0	0	0
3	0.1542	0.3390	0.5013	0.7864	0.9662
4	0	0	0	0	0

Table 6.5: Example failure distributions (50_Y)

ResourceID	50_0.1	50_0.3	50_0.5	50_0.7	50_0.9
1	0	0	0	0	0
2	0.1787	0.3655	0.5352	0.7573	0.9614
3	0.1135	0.3719	0.5884	0.7117	0.9418
4	0	0	0	0	0

6.4.2 Performance Metrics

As a measurement of scheduling performance, we evaluate the following performance metrics:

Scheduling Efficiency: In order to determine the scheduling efficiency, we measure two values of the system: (i) *average makespan per workflow* and (ii) *total number of tasks failed* by all the Grid sites in the system.

The measurement of makespan is taken by averaging over all the workflows in the system. If there are $\boldsymbol{n}$ number of Grid sites and each site has $\boldsymbol{u}$ number of users with each user submitting $\boldsymbol{w}$ number of workflows, then average makespan per workflow in

the system can be defined as,

$$M_{average} = \frac{\sum_{\substack{1 \leq i \leq n \\ 1 \leq j \leq u \\ 1 \leq k \leq w}} M_{i,j,k}}{n \times u \times w}$$

where $M_{i,j,k}$ is the makespan for workflow, $W_{i,j,k}$.

If there are n number of Grid sites and site S_i fails F_i number of tasks, then the *total number of tasks failed* in the system can be defined as,

$$F_{total} = \sum_{1 \leq i \leq n} F_i$$

Scheduling complexity: It is measured as the *total number of task scheduled* by all GASs in the system. If there are n number of Grid sites and GAS_i schedules SCH_i number of tasks, then *total number of task scheduled* in the system can be expressed as,

$$SCH_{total} = \sum_{1 \leq i \leq n} SCH_i$$

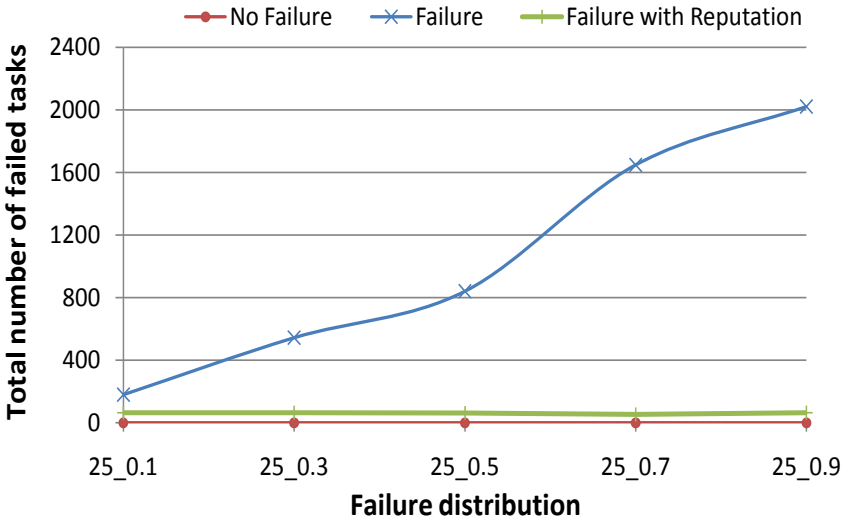
Pruning Efficiency: We consider pruning efficiency as the degree to which the failure-prone resource are shunted out of the system. We have measured *total number of tasks successfully executed and failed* by the resource at each Grid site in order to show the pruning efficiency.

6.4.3 Results and Observations

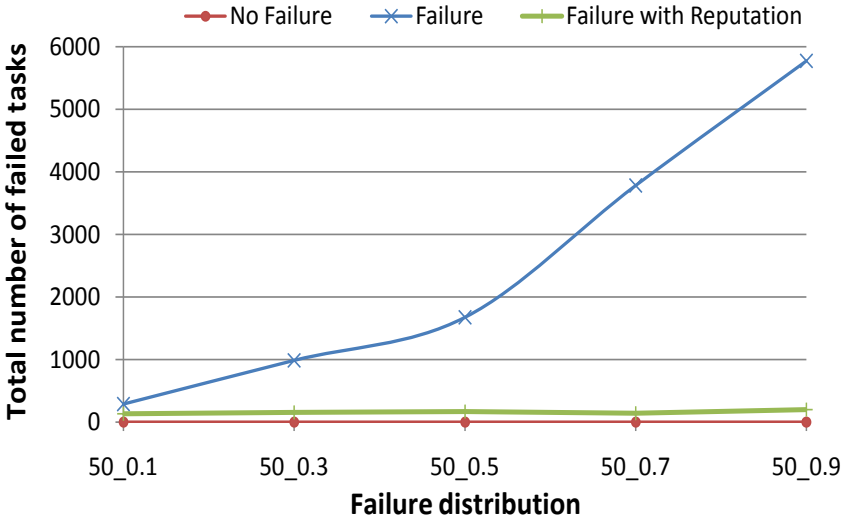
In this section, we present the experimental results obtained by simulating our reputation based dependable workflow scheduling approach and compare these with that of other approaches. The experiments are conducted with the aim at characterizing:

(i) the performance of proposed reputation based dependable scheduling approach (*Failure with Reputation*), compared to its alternatives, *No Failure* (resources do not fail any task) and *Failure without Self-adaptation* (some resources fail tasks and scheduler uses a simple rescheduling technique) with respect to various performance metrics;

(ii) the impact of different resource failure distributions and sizes of workflow on the performance of our approach and of its alternatives;



(a) Total task failures for failure distributions $\mathbf{25\_}\boldsymbol{Y}$



(b) Total task failures for failure distributions $\mathbf{50\_}\boldsymbol{Y}$

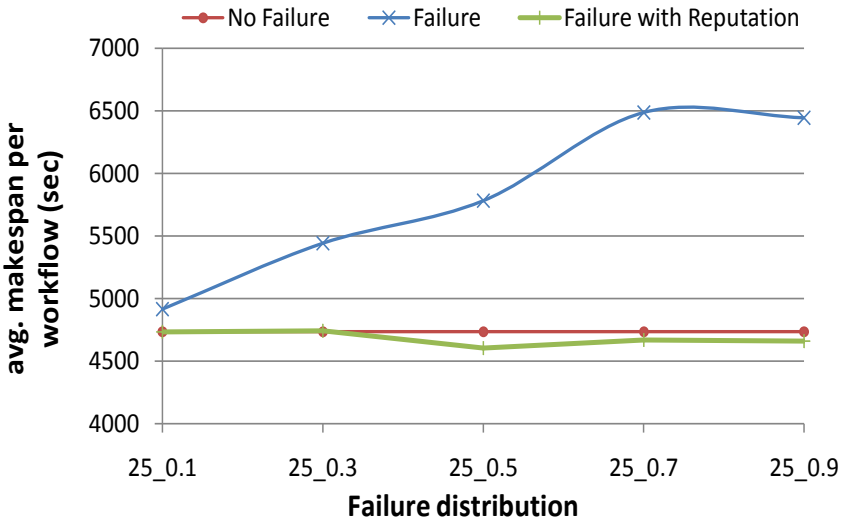
Figure 6.9: Effect of failure distribution on the total number of task failures in the system.

(iii) the significance of reputation threshold ($\boldsymbol{R_{th}}$) on the performance of proposed reputation based scheduling approach.

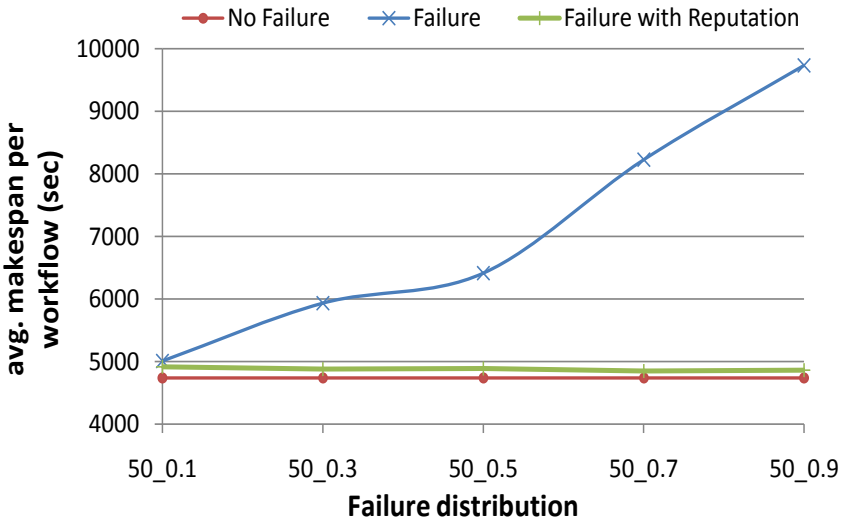
The configuration of different parameters for all the experiments are listed in Table 6.6.

Experiment 1: Measuring Scheduling Efficiency

Experiment 1.1 (Impact of failure distribution): Fig. 6.9 and Fig. 6.10 presents the results of scheduling efficiency of the proposed reputation based scheduling approach



(a) Makespan for failure distributions $\mathbf{25\_}\boldsymbol{Y}$



(b) Makespan for failure distributions $\mathbf{50\_}\boldsymbol{Y}$

Figure 6.10: Effect of failure distribution on the makespan of workflow in the system.

against the other approaches, *Failure without Self-adaptation* and *No Failure*. The total number of tasks failed by all Grid sites, $\boldsymbol{F_{total}}$ for each of the three approaches are depicted in Fig. 6.9(a) and Fig. 6.9(b) for different failure distributions. As we can see from Fig. 6.9(a) that when the failure probability of the resources is increased (for example, from $\mathbf{0.1}$ to $\mathbf{0.9}$), $\boldsymbol{F_{total}}$ in *Failure without Self-adaptation* is heavily increased accordingly.

This situation is further aggravated for $\mathbf{50\_Y}$ (refer to Fig. 6.9(b)) since more resources are likely to fail tasks. In contrast, our approach, *Failure with Reputation* can strongly reduce the number of task failures in the system irrespective of failure distributions. This happens due to the reason that in this case, the resources with higher failure probability are not assigned any task by the schedulers as their reputation scores are decreased beyond the threshold $\boldsymbol{R_{th}}$ after few task failures. Therefore, $\boldsymbol{F_{total}}$ in *Failure with Reputation* is not increased with the increase in failure probability since those failure-prone resources are always shunted out of the system after few failures. For instance, total number of tasks failed by all sites in *Failure with Reputation* is upto $\mathbf{96.8\%}$ and $\mathbf{96.5\%}$ less than that in *Failure without Self-adaptation* for $\mathbf{25\_Y}$ and $\mathbf{50\_Y}$ respectively.

The average makespan per workflow, $\boldsymbol{M_{average}}$ also shows (see Fig. 6.10(a) and Fig. 6.10(b)) similar trend (upto $\mathbf{28\%}$ and $\mathbf{50\%}$ makespan reduction for $\mathbf{25\_Y}$ and $\mathbf{50\_Y}$ respectively) as reflected in total number of task failures since if one task is failed, its child tasks can not be scheduled and eventually the completion time of the whole workflow is increased.

Experiment 1.2 (Impact of number of tasks in workflow): Fig. 6.11 presents the results of scheduling efficiency of the proposed reputation based dependable scheduling approach against the other approaches, *Failure without Self-adaptation* and *Failure* for different sizes of workflow. The results show that if the number of tasks in a workflow increases, $\boldsymbol{M_{average}}$ also increases for all the three approaches since the overall workload on the system is increased. But the impact is more evident for *Failure without Self-adaptation*.

As we can see from Fig.6.11(a) and Fig.6.11(b), in case of *Failure without Self-adaptation*, both $\boldsymbol{F_{total}}$ and $\boldsymbol{M_{average}}$ are increased rapidly with the increase in workflow size (number of tasks). This happens due to the reason that when the workflow size

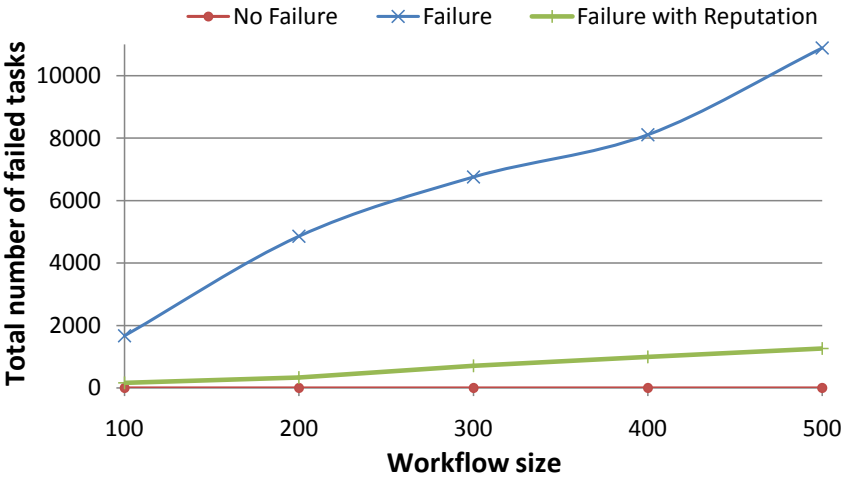
is increased, average number of tasks scheduled per resource in the system is also increased linearly as the number of Grid sites is not changed over time in this experiment. This results in allowing the failure-prone resources to fail more tasks. Thus, $\boldsymbol{F_{total}}$ and $\boldsymbol{M_{average}}$ in *Failure without Self-adaptation* show a piecewise linear growth over the size of workflow.

However, in case of *Failure with Reputation*, when the workload on the system is increased, resources with higher reputation score get more tasks leaving the failure-prone resources isolate. This results in less number of task failures in the system even in higher workload. Therefore, with the increase in workflow size across the system, the performance gain achieved in terms of $\boldsymbol{F_{total}}$ and $\boldsymbol{M_{average}}$ by applying the proposed approach is more evident. For example, when the workflow consists of 500 tasks, $\boldsymbol{F_{total}}$ in *Failure with Reputation* is $\mathbf{88.4\%}$ less than that in *Failure without Self-adaptation* and the makespan reduction is $\mathbf{38.1\%}$ accordingly.

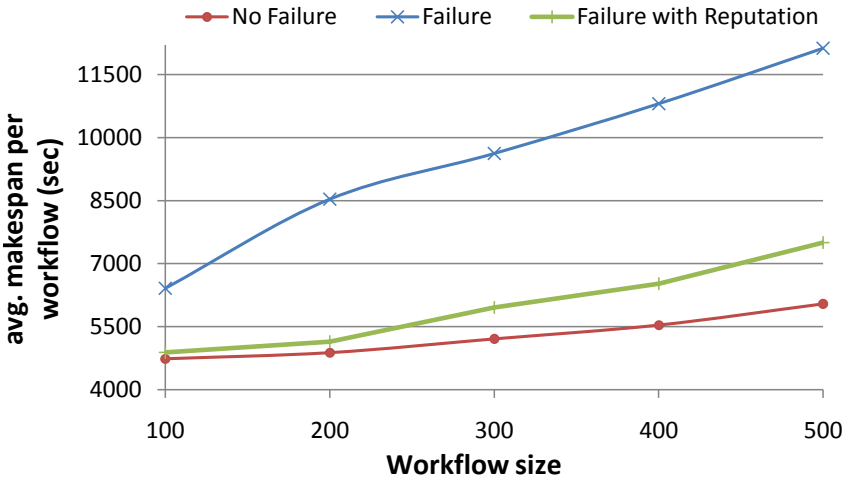
Experiment 2: Measuring Scheduling Complexity

Fig. 6.12 shows the total number of tasks scheduled by GAS1 to GAS16 in the system for the failure distribution, $\mathbf{50\_0.5}$. From the figure, it is evident that in case of *Failure without Self-adaptation*, each GAS needs to schedule more tasks than *No Failure* (where, GAS is not required to schedule any extra task than the size of workflow), which increases the load on the GAS accordingly. On the contrary, in case of *Failure with Reputation*, the number of tasks scheduled by each GAS in the system is almost equal to that of *No Failure* as very few tasks are failed in this approach. For example, GAS14 schedules $\mathbf{100}$ tasks in *No Failure*, $\mathbf{102}$ in *Failure with Reputation*, whereas in case of *Failure without Self-adaptation*, it needs to schedule $\mathbf{216}$ tasks, which is $\mathbf{112\%}$ greater than that in *Failure with Reputation* since $\mathbf{116}$ tasks, scheduled by GAS14 are failed by the Grid sites.

As the other GASs in the system also show the similar trend, the total number of tasks scheduled in the system, $\boldsymbol{SCH_{total}}$ for *Failure with Reputation* ($\mathbf{6569}$) is much smaller than that for *Failure without Self-adaptation* ($\mathbf{8075}$).



(a) Total task failures vs. Number of tasks in workflow



(b) Makespan vs. Number of tasks in workflow

Figure 6.11: Effect of workflow size on F_{total} and $M_{average}$ in the system (failure distribution $50\_0.5$).

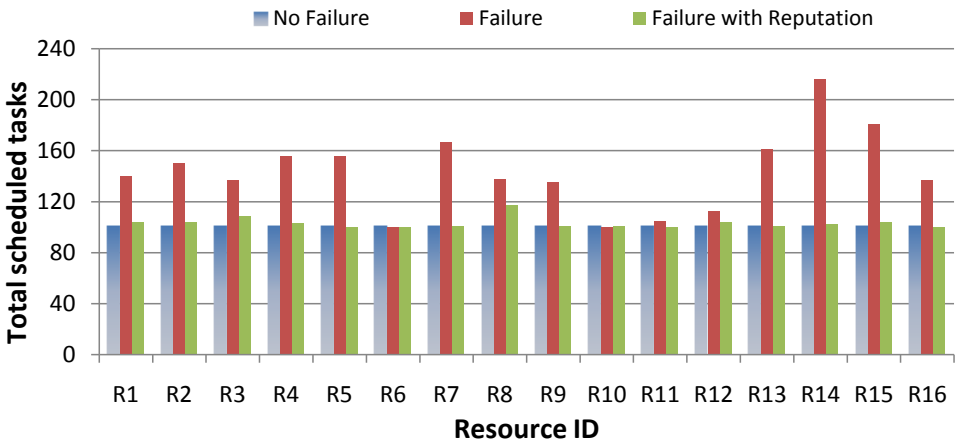


Figure 6.12: Total number of tasks scheduled by the GAS in the system (GAS1 - GAS16) for failure distribution $\mathbf{50\_0.5}$.

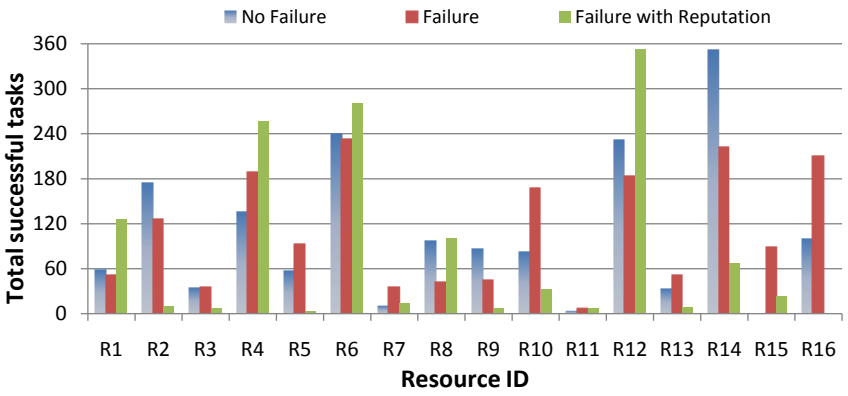
Experiment 3: Measuring Pruning Efficiency

Fig. 6.13 illustrates the pruning efficiency of the proposed scheduling technique. Fig. 6.13(a) and Fig. 6.13(b) show the total number of tasks successfully executed and failed by the resources in Grid site 1 to Grid site 16 respectively for $\mathbf{50\_0.5}$. From the figures, we can realize that in *Failure without Self-adaptation*, if a Grid site can execute task faster, it is assigned more tasks. Thus, the number of successful and failed tasks by that site is high if it's failure probability is low and high respectively.

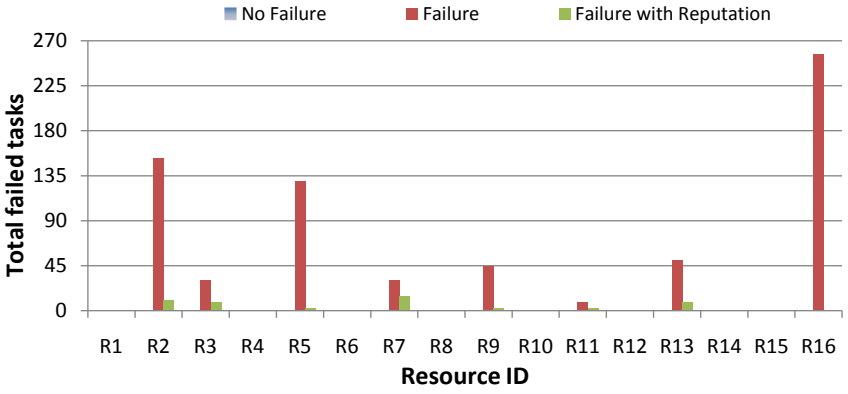
On the other hand, in case of *Failure with Reputation*, number of successful tasks by a Grid site is high if it is faster and does not fail any task. If it fails task, although it can execute task faster, it is not assigned any task further. Therefore, total failed tasks by that resource becomes very low. For instance, total failed tasks by resource R2 (with 0.59 failure probability and 3600 MIPS rating) is $\mathbf{152}$ in *Failure without Self-adaptation*, whereas it is only $\mathbf{11}$ in *Failure with Reputation*. Fig. 6.13(c) shows how failure-prone resource R2 is shunted out of the system over the period of time in our proposed reputation based scheduling approach.

Experiment 4: Impact of Reputation Threshold

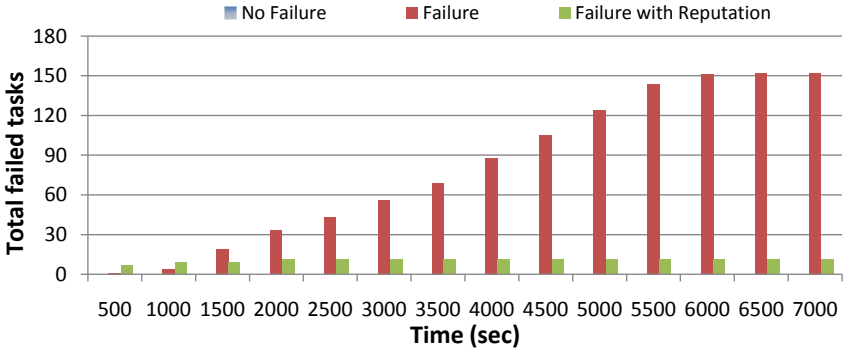
Fig. 6.14 shows the impact of reputation threshold ($\boldsymbol{R_{th}}$) on $\boldsymbol{F_{total}}$ and $\boldsymbol{M_{average}}$ in the system for *Failure with Reputation* when failure distribution is $\mathbf{50\_0.5}$. From the figure, it



(a) Total number of tasks successfully executed by each resource (R1 - R16)



(b) Total number of tasks failed by each resource (R1 - R16)



(c) Total number of tasks failed by Resource **2** over time

Figure 6.13: Effect of considering reputation on pruning failure-prone resources (failure distribution $\mathbf{50\_0.5}$).

Table 6.6: Configuration for different experiments

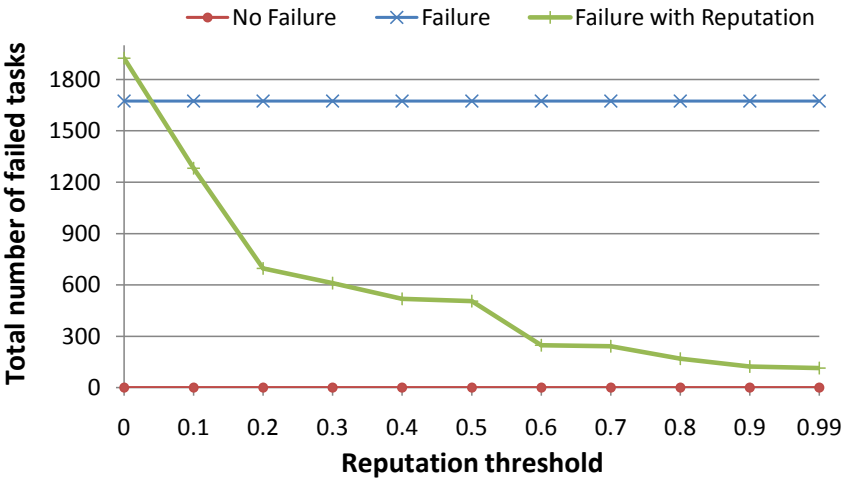
Parameter	Exp 1.1	Exp 1.2	Exp 2	Exp 3	Exp 4	Exp 5
n	64	64	64	64	64	64
No. of tasks	100	100 to 500	100	100	100	100
Task size (MI)	50000 to 500000	50000 to 500000	50000 to 500000	50000 to 500000	50000 to 500000	50000 to 500000
Failure distribution	25_0.1 to 25_0.9, 50_0.1 to 50_0.9	50_0.5	50_0.5	50_0.5	50_0.5	50_0.5
R_{th}	0.8	0.8	0.8	0.8	0.0 to 0.99	0.8
Feedback function	exponential	exponential	exponential	exponential	exponential	simple, exponential

is evident that When R_{th} is slightly higher than 0, F_{total} and $M_{average}$ for *Failure with Reputation* are almost equal to that for *Failure without Self-adaptation*. This happens due to the reason that if R_{th} is very low then the reputation based scheduling scheme is not able to isolate the failure-prone resources with lower reputation score. Hence a considerable amount of tasks are assigned to those resources and F_{total} is increased eventually.

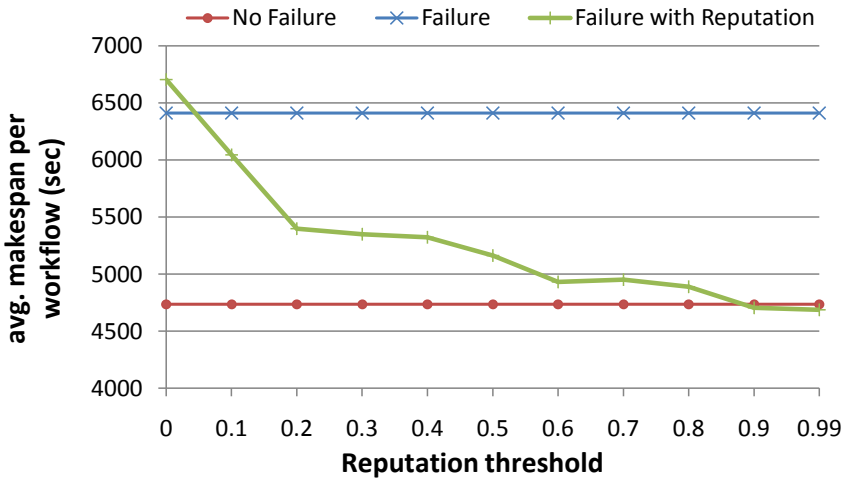
However, when the value of R_{th} is set a little bit higher than 0, performance of the proposed reputation based scheduling approach in terms of scheduling efficiency is improved rapidly. Furthermore, with the increase of the value of R_{th}, average makespan and total task failures for *Failure with Reputation* gradually become almost equal to that for *No Failure*. For example, when R_{th} is set to 0.2, 0.6 and 0.9, F_{total} for *Failure with Reputation* is 58.4%, 85.1% and 92.6% less than that for *Failure without Self-adaptation* respectively. Similarly, $M_{average}$ is also reduced by 15.8%, 23.0% and 26.6% if R_{th} is set to 0.2, 0.6 and 0.9 respectively.

Experiment 5: Performance of Exponential Feedback Function

Fig. 6.15 shows the significance of using exponential feedback functions on F_{total} and $M_{average}$ in the system when the proposed approach, *Failure with Reputation* is employed. In order to measure the performance, we compare our proposed feedback function against a simple linear feedback function available in the literature. From Fig. 6.15(a) and Fig. 6.15(b), it is evident that using an exponential function for calculating feedback results in reduced makespan and less number of total task failures in compare to using a simple linear feedback function. Although $M_{average}$ is not much varied, we can see a



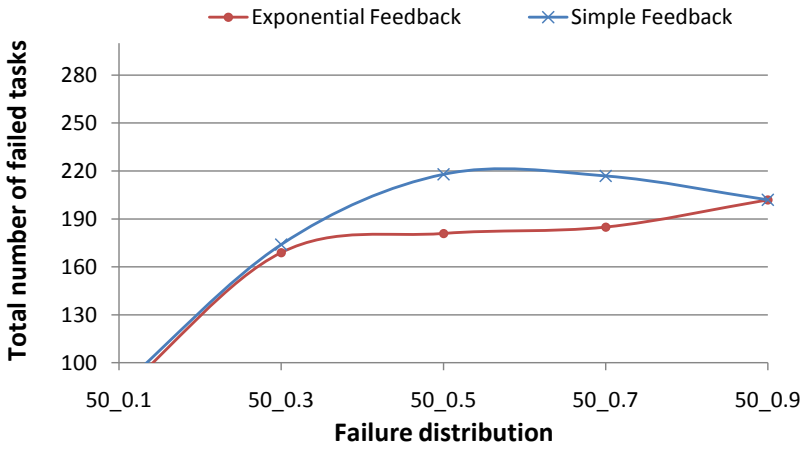
(a) Total task failures vs. Reputation threshold



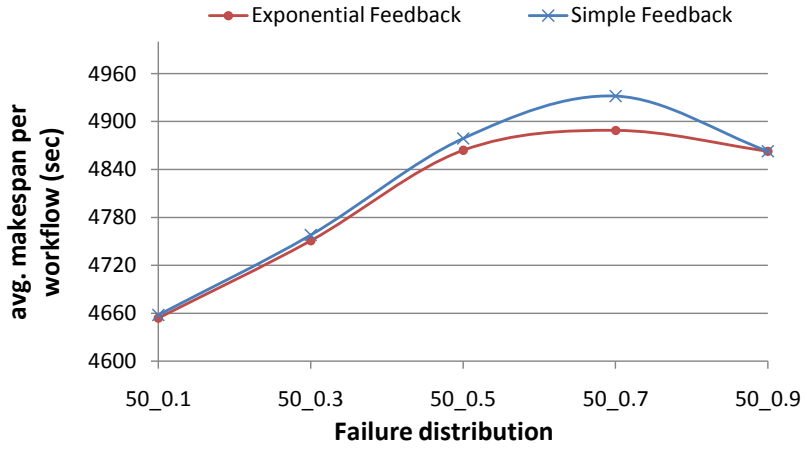
(b) Makespan vs. Reputation threshold

Figure 6.14: Effect of reputation threshold ($\boldsymbol{R_{th}}$) on $\boldsymbol{F_{total}}$ and $\boldsymbol{M_{average}}$ in the system for *Failure with Reputation* (failure distribution $\mathbf{50\_0.5}$).

significant improvement in terms of $\boldsymbol{F_{total}}$. For instance, in case of failure distribution $\mathbf{50\_0.5}$, when simple feedback function is used, $\mathbf{20\%}$ more tasks are failed than using exponential feedback function.



(a) Total task failures

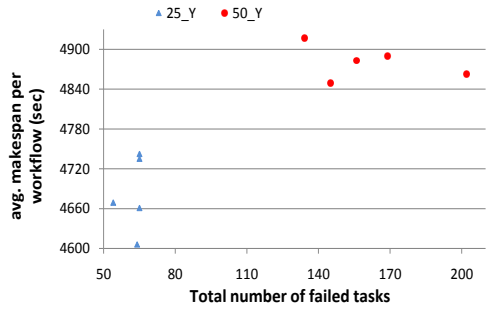


(b) Makespan

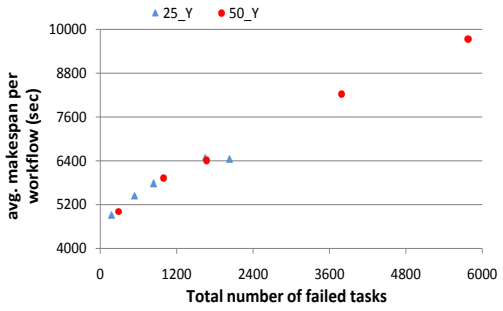
Figure 6.15: Significance of exponential feedback function on $\boldsymbol{F_{total}}$ and $\boldsymbol{M_{average}}$ in the system for *Failure with Reputation* (failure distribution $\mathbf{50\_}\boldsymbol{Y}$).

6.4.4 Discussion and Summary

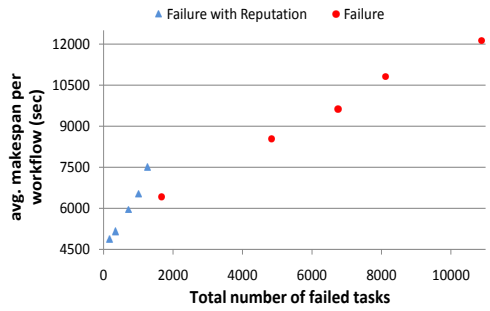
The results from the experiments show that there is a similarity of trend between the two performance metrics $\boldsymbol{M_{average}}$ and $\boldsymbol{F_{total}}$. Thus, we have calculated the *Pearson's correlation coefficient* [15] and plotted the relationship between these metrics in Fig. 6.16.



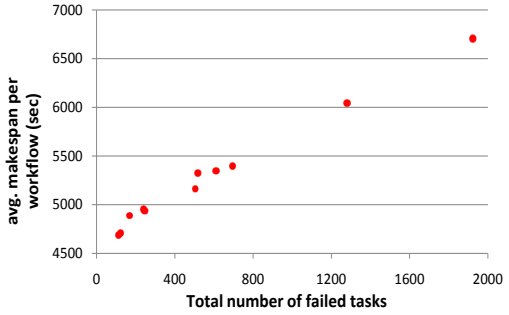
(a) Makespan vs. Total task failures for *Failure with Reputation*



(b) Makespan vs. Total task failures for *Failure without Self-adaptation*



(c) Makespan vs. Total task failures for varying workflow size (failure distribution $\mathbf{50.\_0.5}$)



(d) Makespan vs. Total task failures for varying R_{th} (failure distribution $\mathbf{50.\_0.5}$)

Figure 6.16: Correlation between F_{total} and $M_{average}$ in the system.

Definition 6.2 (Pearson's correlation coefficient): Pearson's correlation coefficient $\rho_{P,Q}$ *between two random variables* P, Q *with means* μ_P, μ_Q *and standard deviations* σ_P, σ_Q *is used to measure the linear relationship between them. It is defined as a quotient of the covariance of the two variables and the product of their standard deviations:*

$$\rho_{P,Q} = \frac{cov(P,Q)}{\sigma_P \sigma_Q} = \frac{E((P-\mu_P)(Q-\mu_Q))}{\sigma_P \sigma_Q}$$

where, $\mu_P = E(P)$ *and* $\sigma_P{}^2 = E(P^2) - E^2(P)$.

The correlation is 1 *when there is a positive linear dependence and* -1 *in case of negative linear dependence. Zero indicates that there is absolutely no linear relationship between the variables.*

The Pearson's correlation coefficient, $\rho_{M_{average},F_{total}}$ between $M_{average}$ and F_{total} in the system for different experiments conducted is listed in Table 6.7. From the table

it can be observed that except for *Failure with Reputation* in *Experiment 1*, the values of $\rho_{M_{average},F_{total}}$ for both *Failure with Reputation* and *Failure without Self-adaptation* are greater than $\mathbf{0.9}$ in all other experiments. This indicates that there is a high degree of positive correlation or linear dependence between $\boldsymbol{M_{average}}$ and $\boldsymbol{F_{total}}$ in the system. This happens due to the reason that when a task is failed, the task that depends on it's output needs to wait for longer period of time to be scheduled and executed. Therefore, the completion time of exit task is delayed and makespan of the workflow is increased, which indicates a linear relationship between $\boldsymbol{M_{average}}$ and $\boldsymbol{F_{total}}$ (see Fig.6.16(b) - Fig.6.16(d)).

However, in case of *Failure with Reputation* in *Experiment 1*, workload is not heavy, $\boldsymbol{F_{total}}$ is small and failure-prone resources are isolated quickly. Thus makespan of the workflow is not increased over the resource failure probability although $\boldsymbol{F_{total}}$ is increase by a small margin. This means that there is no clear relationship or correlation between $\boldsymbol{M_{average}}$ and $\boldsymbol{F_{total}}$ in such situation, which is reflected in Fig. 6.16(a).

Table 6.7: Pearson's correlation coefficient: $\boldsymbol{M_{average}}$ vs. $\boldsymbol{F_{total}}$

Approach	**Exp 1.1 (25_Y)**	**Exp 1.1 (50_Y)**	**Exp 1.2**	**Exp 3**
Failure with Reputation	0.2073	-0.4202	0.9904	0.9382
Failure	0.9673	0.9973	0.9971	-

In summary, the above experimental and analytical studies indicate the following:

(i) considering reputation of Grid sites/resources for scheduling can increase the reliability of application scheduling in Grids and improve the efficiency of distributed schedulers.

(ii) compared to *Failure without Self-adaptation*, *Failure with Reputation* can effectively reduce application completion time by avoiding potential task failures through intelligent scheduling irrespective of failure pattern of resources or workload on the system.

(iii) prunning efficiency of reputation based scheduling approach can be improved by increasing the reputation threshold in the system.

(iv) there is a high degree of positive correlation between makespan of workflow and total task failures in the system.

6.5 Related Work

The main focus of this section is to compare the novelty of the proposed work with respect to existing approaches. We classify the related research into three main areas:

6.5.1 Dependable Scheduling

A recent work by Jik-Soo et al. [69] that advocates Content Addressable Network [105], DHT based dynamic propagation and load-balancing in desktop Grids, suffer from performance uncertainty and unreliability due to the lack of context awareness in scheduling. A most recent proposal on reputation-driven scheduling in the context of voluntary computing environments (desktop grids) has been put forward by Jason et al. [113]. They consider a centralized system model, where a central server is assigned responsibility for maintaining reliability ratings that form the basis for assigning tasks to group of voluntary nodes. Such centralized models for scheduling and reputation management [14][141] present serious bottleneck as regards to scalability of the system and autonomy of Grid sites. Moreover, these approaches are targeted on bag of tasks type of application model, whereas our approach considers scheduling of workflow applications. Currently, Grid information services [34], on which Grid schedulers [47] depend for resource selection, do not provide information regarding how the resources have performed in the recent past (performance history) or at what level they are rated by other schedulers in the system as regards to QoS satisfaction.

6.5.2 Distributed Reputation Models

There has been considerable amount of research work done in P2P reputation systems to evaluate the trustworthiness of participating peers. These reputation systems are targeted towards P2P file sharing networks that focus on sharing and distribution of information in Internet-based environments. The PowerTrust model proposed by Zhou et al. [140], utilizes single dimensional Overlay Hashing Functions (OHFs) for: (i) aassigning score managers for peers in the system and (ii) aggregating/computing the global reputation score. These kinds of OHFs are adequate if the search for peers/resources is based on single keyword (such as file name) or where there is single ordering in search values. How-

ever, OHFs are unable to support (or support with massive overhead) searches containing multiple keywords, range queries (such as search for a Grid site that has: Linux operating system, $\mathbf{100}$ processors, Intel architecture, and reputation $\geq \mathbf{0.5}$). The EigentTrust model [65] suggested by Kamvar et al. also suffers from the shortcomings mentioned above. To overcome these limitations, in the proposed approach a $\boldsymbol{d}$-dimensional data distribution technique [100] is applied on the overlay of peers for managing the information related to complex searches and reputation values.

6.5.3 Grid Workflow Management

With the increasing interest in Grid workflows, many Grid workflow systems, such as Pegasus [36], Triana [118], Taverna [85], Condor DAGMan [76], Kepler [79], SwinDeW-G [131], Gridbus [136] and Askalon [39] have been developed in recent years. Among these systems, in terms of workflow scheduling infrastructure, SwinDeW-G and Triana utilize decentralized P2P based technique. However, the P2P communication in SwinDeW-G and Triana is implemented by JXTA protocol, which uses a broadcast technique. In this work, we use a DHT (such as Chord) based P2P system for handling resource discovery and scheduling coordination. The employment of DHT gives the system the ability to perform deterministic discovery of resources and produce controllable number of messages in comparison to using JXTA.

6.6 Conclusion

In this chapter, we have presented a reputation based dependable scheduling technique that enables self-healing capability for managing workflow applications in global Grids. It proposes a novel algorithm that continuously monitors and calculates the performance (i.e. reliability) of Grid resources based on feedbacks given by individual Grid service consumers and hence is capable of dynamically mapping workflow tasks to the resources that have less likelihood of failure. We have measured the performance of the proposed scheduling technique against two cases: *Failure without Self-adaptation* and *No Failure*. The results show that our scheduling technique can reduce the makespan up to $\mathbf{50\%}$ and successfully isolate the failure-prone resources from the system.

Thus, by applying the proposed reputation based scheduling technique, not only opportunistic and dependable placement of workflow tasks is possible but also significant performance gains are achievable by avoiding potential resource failures. The practical importance of these results is: they highlight the fact that the schedulers, which do not have the ability to self-adapt in dynamic Grid conditions deliver degraded performance to application workflows.

Chapter 7

Conclusions and Future Directions

This thesis addresses the problem of enabling autonomic workflow management for Grid computing to effectively overcome the limitations of inherent uncertainty and dynamism of Grid environment. Autonomic policy based workflow management considers the ability to self-reconfigure to the changes in the Grid environment; discover, diagnose and react to the disruptions of workflow execution as well as monitor and optimize runtime performance automatically. By incorporating the self-* properties of autonomic systems into the workflow management system, we demonstrated that it can effectively overcome the limitations of inherent uncertainty and dynamism of Grid environment as well as significantly improve the performance of workflow execution in global Grids. In this chapter, we first summarize the contributions and findings of this thesis, and then present a list of research issues for future investigations related to this work.

7.1 Summary

Computational Grids provide a global infrastructure for solving large-scale problems in science, engineering, and business. They enable the sharing, selection, and aggregation of geographically distributed heterogeneous resources, such as computational clusters, supercomputers, storage devices, and scientific instruments. Recently, scientific workflows have emerged as important applications for Grid systems; and these workflows are defined, managed, and executed on Grid resources by a specialized middleware, called Workflow Management System (WMS). However, the increasing scale complexity, het-

erogeneity, and dynamism of Grid environment have made such WMS brittle, unmanageable, and insecure. This thesis have presented a framework for effectively managing workflow applications in global Grids considering this problem by the following key contributions:

- proposed a taxonomy of autonomic application management for Grid computing and surveyed existing Grid system (specially, workflow management systems).
- developed a dynamic critical path based workflow scheduling algorithm that can adapt to changing Grid environment.
- designed an architecture for autonomic workflow management system in Grids according to the requirements identified in proposed taxonomy.
- devised a decentralized and cooperative workflow scheduling algorithm utilizing a self-configuring P2P overlay structure with regards to resource discovery, coordination, and overall system decentralization.
- leveraging the proposed autonomic workflow management architecture, developed a reputation-based dependable workflow scheduling technique to enable self-healing behavior in Grid workflow management system.

Next, we discuss the key findings of this thesis. In Chapter 2, we provided a taxonomy of autonomic application management in Grids and a detailed survey of several representative Grid systems (specially, workflow management systems) to demonstrate the comprehensiveness of the taxonomy. From the taxonomy and survey, we identified that most of the existing Grid workflow management systems are centralized and do not support cooperative application scheduling. In addition, as these Grid systems are highly complex and volatile, most of them incorporates self-optimizing and self-healing properties of an autonomic computing system. However, in order to cope up with the increasing-scale complexity and volatility of Grid environment, these systems are also required to address the self-configuring and self-protecting policies to some extent.

In Chapter 3, we proposed a dynamic critical path based workflow scheduling heuristic (DCP-G) that takes into account the dynamic behavior of Grid resources. This chapter

identified that dynamic scheduling approaches can adapt to temporal behaviour of heterogeneous Grid resources and are able to avoid performance degradation by generating efficient schedules. The main findings of this chapter are: (i) scheduling time for meta-heuristic based workflow scheduling techniques increases with the size of the workflow and is much higher than that of heuristic based techniques; (ii) meta-heuristic based techniques can generate more effective schedule than heuristic based techniques only in static environment, but they suffer from the problem of higher scheduling time overhead; and (iii) in dynamic environment, where resource availability changes frequently, DCP-G can generate better schedule than that of other approaches irrespective of workflow type and size.

In Chapter 4, we described the architecture of an autonomic workflow management system, designed according to the requirements identified in the taxonomy presented in Chapter 2. In particular, we discussed the system models, such as Grid model, coordination model, and application model that form the basis of this architectural framework. Moreover, the prototype implementation of the proposed architecture is also provided in this chapter. The prototype wokflow management system is developed by leveraging Aneka enterprise Grid platform. Further, the required components and services of the prototype system, such as application development, resource discovery, coordination, and task scheduling are also illustrated with directions for deployment of the system.

In Chapter 5, we proposed a decentralized and cooperative workflow scheduling approach that leverages a DHT-based self-configuring overlay for resource coordination and a cooperative decision making strategy to achieve runtime optimization. This chapter identified that decentralization of resource organization through self-configuring P2P overlay can facilitate to enable autonomic management of workflow applications in Grids with respect to resource configuration and information management. The main findings of this chapter are: (i) proposed scheduling technique is scalable with respect to scheduling message complexity; (ii) for decentralized and cooperative workflow scheduling, average response time of a task and makespan of the fork-join workflows in the system are linearly dependant on the number of tasks in a workflow; (iii) cooperative scheduling technique can not only reduce the makespan of workflow by decreasing the task waiting time but also balance the load among the available Grid resources significantly; and (iv) decentral-

ized coordination technique is successful in generating schedules, which are as efficient as the centralized coordination approach that has global knowledge of the system.

Finally, in Chapter 6, we utilized the self-configuring and self-optimizing nature of this decentralized and cooperative scheduling technique, presented in Chapter 5 to propose a reputation based dependable workflow scheduling technique that enables self-healing property for the workflow management system to exhibit autonomic behaviour. The key findings are: (i) considering reputation of Grid sites/resources for scheduling can increase the reliability of application scheduling in Grids and improve the efficiency of distributed schedulers; (ii) proposed scheduling approach can effectively reduce application completion time by avoiding potential task failures through intelligent scheduling irrespective of the failure pattern of resources or workload on the system; (iii) proposed approach can successfully isolate failure-prone resources form the system; and (iv) for reputation based workflow scheduling, there is a high degree of positive correlation between makespan of workflow and total task failures in the system.

To summarize, this thesis has laid the foundation for autonomic workflow management system for Grids with a novel suit of architectural framework and scheduling algorithms in order to facilitate the workflow scheduler to incorporate the autonomic computing properties and features so that it can adapt to the changes in Grid environment efficiently and optimize its performance. With these novel contributions, this thesis opens up opportunities for future research in relation to managing the uncertainty and dynamism in distributed computing systems and computational Clouds.

7.2 Future Directions

This thesis improves the understanding of autonomic management of workflow applications in Grid computing environment and advances the state-of-the-art through its contributions. The investigations conducted in this thesis reveal several areas in Grid workflow management, where much work is remained to be done. Moreover, the contributions of this thesis have led to new challenges that are required to be addressed through further research. This section briefly describes some of these challenges within the scope of the thesis.

7.2.1 Workflow Management for Data-intensive Applications

Workflow applications in present-day Grids are usually divided into two categories with regards to the dependencies among the tasks: compute-intensive and data-intensive. In a compute-intensive workflow, the tasks in a workflow are ordered based on the control dependency among them. Thus, the data flow among the tasks is generally in the range of few KiloBytes (KB) to MegaBytes (MB). In contrast, the data flow among the tasks in a data-intensive workflow is in the range of hundreds of MegaBytes to PetaBytes (PB). In other words, the requirements of resource inter-connection bandwidth for transferring data in a data-intensive workflow outweigh the computational requirements for processing tasks. This, as a consequence, demands more time to transfer and store data as compared to execute tasks of a workflow.

Moreover, managing data-intensive workflow applications in Grids also needs to handle different types of data, such as input data, backend databases, intermediate data products, and output data. Many Bioinformatics applications [87] often have small input and output data but rely on massive backend databases that are queried as part of task execution. On the other hand, some Astronomy applications [73] generate huge output data that are feed into other applications for further processing. Some applications also need the data to be streamed between the tasks for efficient execution.

In this thesis, we model scientific applications as compute-intensive workflows, where data dependency among the tasks in workflow is negligible. Thus, research initiatives are needed to be directed towards autonomic management of data-intensive workflow applications in Grids, addressing the challenges with regards to handling huge data, such as parallel data transfer, streaming, replica management, and optimal utilization of available network capacity. In Chapter 4, we propose a coordination model to manage the coordination objects (i.e. resource claim and resource ticket) for coordinated resource provisioning. These coordination objects support provisioning of computational resources with a view to schedule compute-intensive workflows. Therefore, an immediate follow-up work would be to extend this framework to support provisioning of storage resources with the focus of scheduling data-intensive workflow applications in Grids.

7.2.2 Enhancing Reliability of Critical Tasks

In Grid environment, execution of a task in workflow is generally failed for various reasons, such as changes in execution environment configuration, non-availability of required services or software components, and faults in computational and network fabric components. A number of techniques (i.e. retry, check-pointing, and redundant task-allocation) have been proposed to achieve fault-tolerance in workflow management [10] as discussed in Chapter 2. The redundant task-allocation technique [10] executes the same task simultaneously on different Grid resources to guarantee fault-tolerant execution of that task in the event of task failure, provided that one of the resources does not fail.

Due to the scarcity of computational resources and robustness of workflow applications, its not efficient to apply redundant task-allocation technique for each task in a workflow, rather it can be applied for the *critical tasks*. A task in a workflow is called *critical task* if the execution of multiple tasks depend on the output or the execution completion of that task. The critical task replication mechanism replicates a critical task execution on more than one resource. The result, produced earliest from these tasks is then used for the rest of the workflow. For example, the workflow submitted by $\boldsymbol{user1}$ in Fig. 7.1 has two critical tasks, $\boldsymbol{T_1}$ and $\boldsymbol{T_4}$. Thus, $\boldsymbol{GAS1}$ submits two resource claim requests for $\boldsymbol{T_1}$ so that $\boldsymbol{T_1}$ can be executed on multiple resources, and if one resource fails, the result/output generated in other resource can be used to start execution of its child tasks, $\boldsymbol{T_2}$ and $\boldsymbol{T_3}$.

Therefore, in addition to utilizing the reputation based scheduling technique proposed in Chapter 6, redundant task-allocation technique can also be leveraged to further explore the research theme endeavoured in this thesis, i.e. improving reliability of workflow execution in case of temporal resource behaviour in Grid environment. The challenges to be tackled along this research direction include developing algorithms for identifying the critical tasks and determining the level of redundancy based on the reliability requirement. Furthermore, once output of the successfully executed critical task is transferred as input for the execution of its child tasks, the redundant execution requests for this critical task that are already queued or under processing are required to be terminated for preventing unnecessary consumption of computing power in the system. Efficient Algorithms should be developed to effectively resolve this issue and optimize resource consumption.

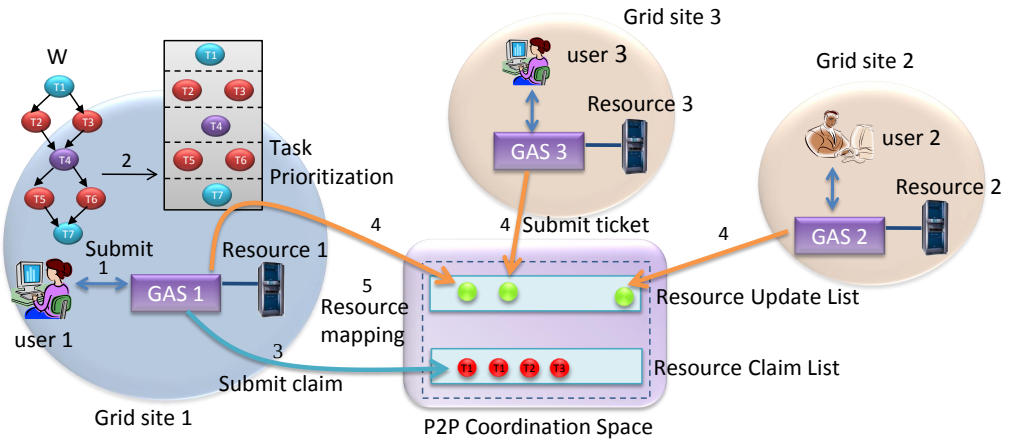


Figure 7.1: Enhancing reliability of execution for critical tasks in workflow by redundancy.

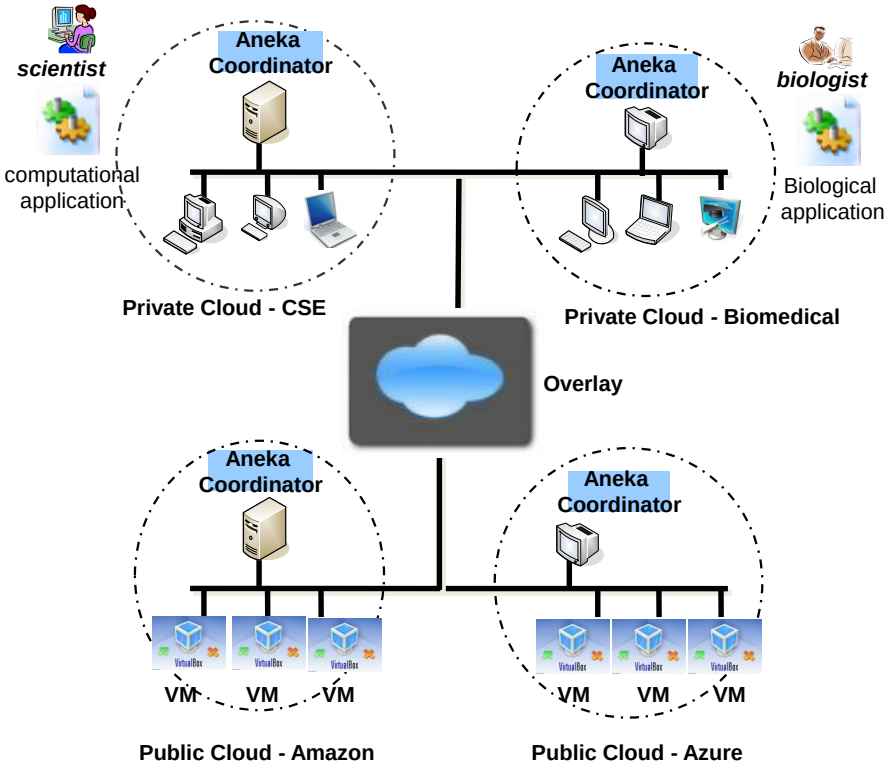


Figure 7.2: Autonomic workflow management in Aneka Enterprise Cloud platform by leveraging public and private Cloud services.

7.2.3 Self-protection

Self-protecting refers to the ability to anticipate and protect against threats or intrusions. This property makes an autonomic system capable of detecting and protecting itself from malicious attacks so as to maintain overall system security and integrity. Self-protecting system management can be achieved by implementing some proactive policies (i.e. dynamic access control) at both resource and user sides, such as providing accurate warning about potential malicious attack, taking networked resources offline if any anomaly is detected, and shutting down the system if any hazardous event occurs that can possibly damage the system.

The focus of this thesis is to enhance the performance or reliability of workflow execution in dynamic Grid environment by designing an autonomic workflow management system with the incorporation of fundamental aspects of an autonomic system: four self-* properties. However, in this thesis, we mainly discuss on enabling self-configuring, self-optimizing, and self-healing properties. Thus, to further improve the proposed framework for autonomic workflow management system, algorithms and policies for self-protection are required to be developed. In this regard, one research direction would be to explore the utilization of dynamic key management scheme, where keys are pre-distributed and dynamically updated by periodic or ten re-keying operations. Detailed information on key pre-distribution, revocation, and dynamic updating schemes for active key management is available in existing literature [25] [77].

Another interesting approach to protect the system from malicious attacks is to leverage intrusion detection techniques [56], where the system performs online monitoring and analyzes the attacks/intrusions. Then a model is devised and trained using the past data that is used to successfully and efficiently detect future attacks. The research challenges in this direction are: (i) developing a intrusion detection model for Grid resources; (ii) dynamically updating or training the model with online monitoring data; and (iii) integrating the model into GAS for realtime detection and prevention.

7.2.4 Autonomic Workflow Management in Clouds

Cloud Computing [23] has emerged as the next generation platform for hosting business and scientific applications. It offers infrastructure, platform, and software as services that are made available as on-demand and subscription-based services in a pay-as-you-go model to users. The key characteristics of Cloud include scalability and reliability, which ensure the consumers of Cloud services that the Cloud infrastructure is very robust and will always be available at any time.

With regards to deployment, there are mainly two types of Clouds: (i) public Cloud (e.g. Amazon EC2 [123]), where resources are dynamically provisioned on a fine-grained self-service basis over the Internet via web services; and (ii) private Cloud, where resources are made available to a limited number of service consumers within the organization behind a firewall with more control over their data. Now, in order to inter-connect or combine these private and public Clouds for creating hybrid Cloud environment, utilization of P2P routing and information dissemination structure is essential to avoid the problems of provisioning efficiency bottleneck and single point of failure that are predominantly associated with traditional centralized or hierarchical approaches.

Thus, the DHT-based self-configuring P2P overlay, proposed in this thesis for autonomic management of workflow applications can be utilized to support scalable and self-managing service discovery and load-balancing in Cloud computing environments. In Chapter 4, we discuss prototype implementation of the self-managing workflow management system utilizing Aneka Enterprise Grid platform. This work can be further extended to develop Cloud application management and resource provisioning system using Aneka Enterprise Cloud platform. As shown in Fig. 7.2, the Aneka Coordinator can be deployed for each Cloud site (private or public) containing multiple resources (i.e. VMs) and hosted a Cloud peer service that essentially glues different Cloud services to the sefl-configuring P2P overlay.

7.2.5 Data Analytics Workflow Scheduling in Hybrid Clouds

Cloud computing environments are not only dynamic but also heterogeneous with multiple types of services (e.g. infrastructure, platform, and software) offered by various

service providers (e.g. Amazon). Scheduling data analytics workflow applications in such environment (refer to Fig. 7.3 and Fig. 7.4) requires to address a number of issues, including minimizing cost and time of execution, satisfying user's QoS constraints, and considering the temporal behavior of the environment. The majority of scheduling techniques [88][130] proposed to solve these issues are based on meta-heuristics, which produce a good schedule given the current state of Cloud services and reserve the services in advance accordingly.

On the other hand, the existing heuristic based scheduling techniques [78][95] as discussed in Chapter 3 are dynamic in nature and map the workflow tasks to services on-the-fly, but lack the ability of generating schedule considering workflow-level optimization and user QoS constraints, such as deadline and budget. Moreover, most of these techniques do not consider the data placement constraints imposed by the Cloud users, while scheduling the data-analytics workflows.

Thus, it is necessary to develop a hybrid heuristic that can effectively integrate most of the benefits of existing approaches to optimize execution cost and time as well as meet the user's requirements through an adaptive fashion in order to efficiently manage the dynamism and heterogeneity of the hybrid Cloud environment.

7.2.6 Energy-aware Autonomic Resource Allocation

The growing demand for computational power from industry and academia to execute scientific, business and web applications has lead to excessive electrical power consumption. Although a number of initiatives have been taken recently that resulted in development of energy-efficient hardware solutions, the overall energy consumption continues to grow due to the overwhelming requirements for computing resources and data centers. For example, the cost of energy consumption by IT infrastructure in USA was estimated as 4.5 billion dollars in 2006, which is likely to be doubled by 2011 [9]. This phenomena of high power consumption is eventually leading to some critical problems (e.g. insufficient or malfunctioning cooling system may result in overheating of the resources reducing system reliability and devices lifetime). Moreover, this high power consuming infrastructure is generating substantial amount of carbon dioxide (CO2) emissions that is contributing

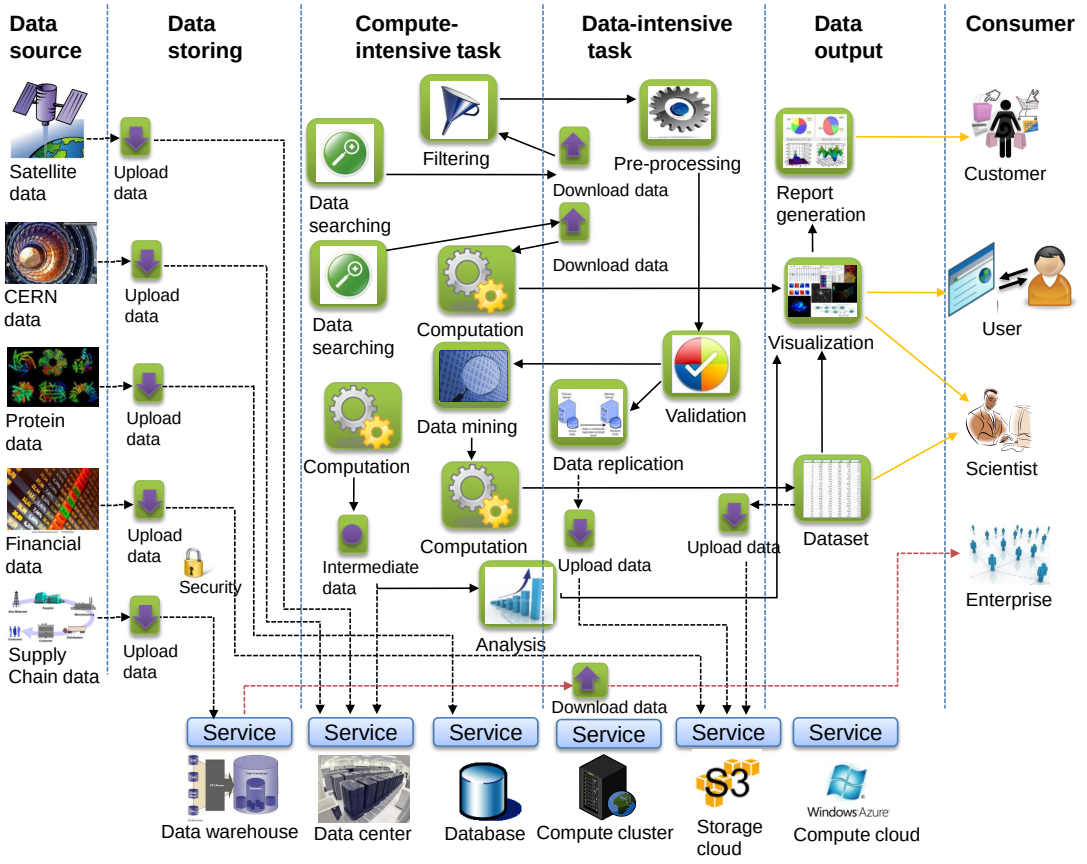


Figure 7.3: An example of data analytics workflow execution in Cloud.

to the greenhouse effect.

As Grids consist of distributed heterogeneous resources, such as computational clusters and supercomputers, energy efficient resource allocation and application management in Grids has been emerged as a prominent area of research nowadays. Many techniques have already been applied to reduce power usage within the resource site: switching off parts of the cluster that are not utilized [127] or Dynamic Voltage Scaling (DVS) to slow down the speed of CPU processing [111]. Another approach to reduce system-wide energy consumption is to distribute the compute-intensive parallel applications considering the energy status of Grid resources [70].

In Chapter 4, we propose an architectural framework for autonomic management of workflow applications. One of the key component of this framework is Grid Autonomic Manager (GAM) that is responsible for scheduling jobs (workflows) submitted by the users in the system. The proposed GAM does not take into account power consumption

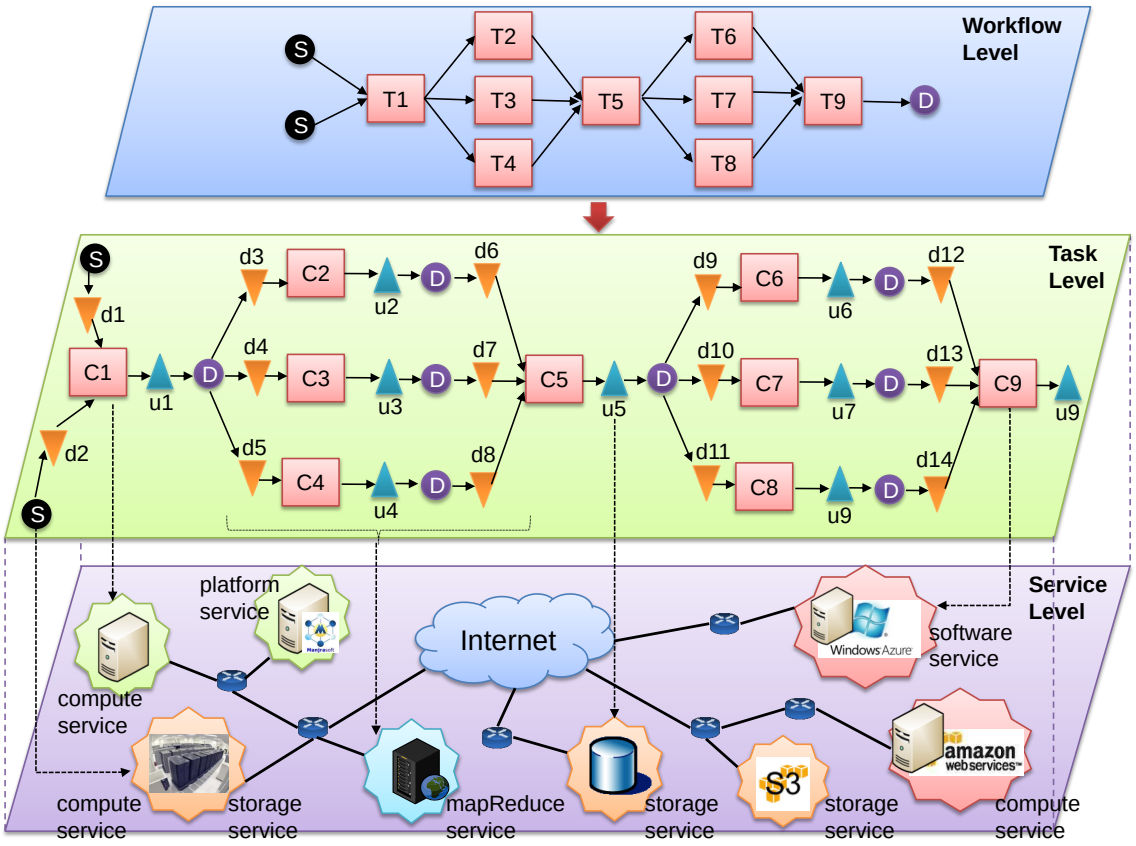


Figure 7.4: Layered architecture for workflow execution in hybrid Cloud.

for making scheduling decisions. Thus, one further direction of the research work presented in this thesis would be to extend this GAM with energy-aware resource allocation algorithms that take into account both consolidation (to switch off nodes) and smart task mapping techniques with a view to lower the total energy consumed to run an application. It can be achieved by two ways: firstly, allocating the tasks in a workflow to resources, where it induces lower power consumption; secondly, allocating the tasks in such a way that the number of switched on hosts is reduced, considering the fact that most of the power of an host is consumed when it is switched on.

REFERENCES

[1] Better business using grid solutions, bingrid. http://www.beingrid.eu [august 2010].

[2] The directed acyclic graph manager, condor project. http://www.cs.wisc.edu/condor/dagman/ [august 2010].

[3] Dynamic assembly for system adaptability, dependability, and assurance. http://web.cs.wpi.edu/~heineman/dasada/ [august 2010].

[4] Extensible markup language (xml) 1.0 (third edition). http://www.w3.org/TR/REC-xml/ [august 2010].

[5] Griphyn: Grid physics network project. *http://www.griphyn.org. [August 2010].*

[6] Situational awareness system. http://www.darpa.mil/sto/strategic/suosas.html [august 2010].

[7] Web services description language (wsdl) version 1.2. http://www.w3.org/TR/wsdl12 [august 2010].

[8] World wide web consortium (w3c). http://www.w3.org/ [august 2010].

[9] Epa report to congress on server and data center energy efficiency: Public law 109-431, u.s. environmental protection agency energy star program. Technical report, Lawrence Berkeley National Laboratory, USA, August, 2007.

[10] J. H. Abawajy. Fault-tolerant scheduling policy for grid computing systems. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS04), USA*, April, 2004.

[11] M. Agarwal, V. Bhat, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, L. Zhen, M. Parashar, B. Khargharia, and S. Hariri. Automate: Enabling autonomic applications on the grid. In *Proceedings of the Autonomic Computing Workshop, USA*, June, 2003.

[12] S. Agarwala, Y. Chen, D. S. Milojicic, and K. Schwan. Qmon: Qos- and utility-aware monitoring in enterprise systems. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC'06), Ireland*, June, 2006.

[13] A. Auyoung, B. Chun, A. Snoeren, and A. Vahdat. Resource allocation in federated distributed computing infrastructures. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-demand IT Infrastructure (OASIS'04), USA*, October, 2004.

[14] F. Azzedin and M. Maheswaran. Integrating trust into grid resource management systems. In *Proceedings of the 31st International Conference on Parallel Processing (ICPP'02), Canada*, August, 2002.

[15] N. Balakrishnan and C. R. Rao. Order statistics: Applications. *Handbook of Statistics, vol. 17, Elsevier Science Pub Co*, 1998.

[16] W. H. Bell, D. G. Cameron, L. Capozza, A. P. Millar, K. Stockinger, and F. Zini. Simulation of dynamic grid replication strategies in optorsim. In *Proceedings of the 3rd IEEE/ACM International Workshop on Grid Computing (Grid'02), USA*, November, 2002.

[17] V. Bhat, V. Matossian, M. Parashar, M. Peszynska, M. K. Sen, P. L. Stoffa, and M. F. Wheeler. Autonomic oil reservoir optimization on the grid. *Concurrency and Computation: Practice and Experience, vol. 17, no. 1, pp. 1-26*, 2005.

[18] P. Blaha, K. Schwarz, G. Madsen, D. Kvasnicka, and J. Luitz. Wien2k - an augmented plane wave plus local orbitals program for calculating crystal properties. Technical report, Vienna University of Technology, Austria, 2001.

[19] J. Blythe, S. Jain, E. Deelman, A. Gil, and K. Vahi. Task scheduling strategies for workflow-based applications in grids. In *Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05), UK*, May, 2005.

[20] B. Bode, D. Halstead, R. Kendall, and D. Jackson. Pbs: The portable batch scheduler and the maui scheduler on linux clusters. *Proceedings of the 4th Linux Showcase and Conference, USA*, October, 2000.

[21] 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'00), China*, May, 2000.

[22] R. Buyya and M. Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience, vol. 14, no. 13-15, pp. 1175-1220*, 2002.

[23] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems, vol. 25, no. 6, pp. 599-616*, 2009.

[24] E. Byun, Y. Kee, E. Deelman, K. Vahi, G. Mehta, and J. Kim. Estimating resource needs for time-constrained workflows. In *Proceedings of the 4th IEEE International Conference on eScience (eScience'08), USA*, December, 2008.

[25] H. Chan, V. D. Gligor, A. Perrig, and G. Muralidharan. On the distribution and revocation of cryptographic keys in sensor networks. *IEEE Transactions on Dependable and Secure Computing, vol. 2, no. 3, pp. 233-247*, 2005.

[26] L. Chan and S. Karunasekera. Designing configurable publish-subscribe scheme for decentralised overlay networks. In *Proceedings of the 21st IEEE International Conference on Advanced Information Networking and Applications (AINA'07), Canada*, May, 2007.

[27] S. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw. The legion resource management system. In *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'99) in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS'99), Puerto Rico*, April, 1999.

[28] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Germany*, August, 2003.

[29] G. Chen, C. P. Low, and Z. Yang. Coordinated service provisioning in peer-to-peer environments. *IEEE Transactions on Parallel and Distributed Systems, vol. 19, no. 4, pp. 433-446*, 2008.

[30] D. Chiu, S. Deshpande, G. Agrawal, and R. Li. Cost and accuracy sensitive dynamic workflow composition over grid environments. In *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (Grid'08), Japan*, September, 2008.

[31] W. Cirne, F. Brasileiro, J. Sauve, N. Andrade, D. Paranhos, E. Santos-Neto, and R. Medeiros. Grid computing for bag of tasks applications. In *Proceedings of the 3rd IFIP Conference on E-Commerce, E-Business and E-Government, Brazil*, September, 2003.

[32] L. Clementi, S. Krishnan, W. Goodman, J. Ren, W. W. Li, P. W. Arzberger, G. Vareille, S. Dallakyan, and M. F. Sanner. Services oriented architecture for managing workflows of avian flu grid. In *Proceedings of the 4th IEEE International Conference on eScience (eScience'08), USA*, December, 2008.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, USA, 2nd edition, 2001.

[34] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC'01), USA*, June, 2001.

[35] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03), USA*, February, 2003.

[36] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflow onto the grid. In *Proceedings of the 2nd European Across Grids Conference (AxGrids'04), Cyprus*, January, 2004.

[37] E. Deelman, G. Singh, M. 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, vol. 13, no. 3, pp. 219-237*, 2005.

[38] P. Durschel. The renaissance of decentralized systems. In *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC'06), France*, June, 2006.

[39] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto, and H. L. Truong. Askalon: A tool set for cluster and grid computing. *Concurrency and Computation: Practice and Experience, vol. 17, no. 2-4, pp. 143-169*, 2005.

[40] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with agwl: An abstract grid workflow language. In *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'05), UK*, May, 2005.

[41] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC'97), USA*, August, 1997.

[42] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications, vol. 11, no. 2, pp. 115-128*, 1997.

[43] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, USA, 1999.

[44] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS'98), USA*, November, 1998.

[45] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications, vol. 15, no. 3, pp. 200222*, 2001.

[46] I. Foster, J. Vckler, M. Wilde, and Y. Zhao. Chimera: A virtural data system for representing, querying, and automating data derivation. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management (SSDBM'02),Scotland*, July, 2002.

[47] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-g: A computation management agent for multi-institutional grids. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC'01), USA*, June, 2001.

[48] A. Ganek and T. Corbi. The drawing of the autonomic computing era. *IBM Systems Journal, Special Issue on Autonomic Computing, vol. 24, no. 1, pp. 5-18*, 2003.

[49] P. Garca, C. Pairot, R. Mondjar, J. Pujol, H. Tejedor, and R. Rallo. Planetsim: A new overlay network simulation framework. In *Proceedings of the 4th International Workshop on Software Engineering and Middleware (SEM'04), Austria*, September, 2004.

[50] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.

[51] W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Proceedings of the 1st IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01), Australia*, May, 2001.

[52] Y. Gil, V. Ratnakar, E. Deelman, G. Mehta, and J. Kim. Wings for pegasus: Creating large-scale scientific applications using semantic representations of computational workflows. In *Proceedings of the 19th National Conference on Innovative Applications of Artificial Intelligence (IAAI07), Canada*, July, 2007.

[53] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, USA, 1989.

[54] L. Gong. JXTA: A network programming environment. *IEEE Internet Computing, vol. 5, no. 3, pp. 88-95*, 2001.

[55] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: Content-based publish/subscribe over p2p networks. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware (Middleware'04), Canada*, October, 2004.

[56] K. K. Gupta, B. Nath, and K. Ramamohanarao. Layered approach using conditional random fields for intrusion detection. *IEEE Transactions on Dependable and Secure Computing, vol. 7, no. 1, pp. 35-49*, 2010.

[57] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. Evaluation of job-scheduling strategies for grid computing. In *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (Grid'00), India*, December, 2000.

[58] P. Horn. Autonomic computing: Ibm's perspective on the state of information technology. Technical report, IBM Corporation, October, 2001.

[59] M. C. Huebscher and J. A. McCann. A survey of autonomic computing degrees, models, and applications. *ACM Computing Surveys, vol. 40, no. 3, pp. 1-28*, 2008.

[60] IBM-Corporation. A technical view of autonomic computing. *Software Group, IBM USA*, 2002.

[61] IBM-Corporation. An architectural blueprint for autonomic computing. Technical report, 2003.

[62] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. Epema. The grid workloads archive. *Future Generation Computer Systems, vol. 24, no. 7, pp. 672-686*, 2008.

[63] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. Epema. The grid workloads archive. *Future Generation Computing Systems*, 2009.

[64] M. B. Juric, B. Mathew, and P. Sarang. *Business Process Execution Language for Web Services*. Packt Publishing, UK, 2004.

[65] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th International Conference on World Wide Web (WWW'03), Hungary*, May, 2003.

[66] D. Kececioglu. *Reliability Engineering Handbook, vol. 1*. Prentice Hall, USA, 1991.

[67] H. Kim and M. Parashar. *CometCloud: An autonomic cloud engine, Cloud Computing: Principles and Paradigms, R. Buyya et al. (eds.)*. Wiley Press, USA, 2011.

[68] H. Kim, M. Parashar, D. J. Foran, and L. Yang. Investigating the use of autonomic cloudbursts for high-throughput medical image registration. In *Proceedings of the 10th IEEE/ACM International Conference on Grid Computing (Grid'09), Canada*, October, 2009.

[69] J. Kim, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman. Using content-addressable networks for load balancing in desktop grids. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC'07), USA*, June, 2007.

[70] K. Kim, R. Buyya, and J. Kim. Power aware scheduling of bag-of-tasks applications with deadline constraints on dvs-enabled clusters. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07), Brazil*, May, 2007.

[71] S. Kim and J. Browne. A general approach to mapping of parallel computation upon multiprocessor architectures. In *Proceedings of the 27th IEEE International Conference on Parallel Processing (ICPP'98), USA*, August, 1988.

[72] Y. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 7, pp. 506-521*, 1996.

[73] A. C. Laity, N. Anagnostou, G. B. Berriman, J. C. Good, J. C. Jacob, D. S. Katz, and T. Prince. Montage: An astronomical image mosaic service for the nvo. In *Proceedings of the 14th Annual Conference on Astronomical Data Analysis Software and Systems (ADASS'XIV), USA*, October, 2004.

[74] E. Laure and B. Jones. Enabling grids for e-science: The egee project. Technical Report EGEE-PUB-2009-001, CERN, Switzerland, 2009.

[75] Z. Li and M. Parashar. Comet: A scalable coordination space for decentralized distributed environments. In *Proceedings of the 2nd International Workshop on Hot Topics in Peer-to-Peer Systems (HOT-P2P'05), USA*, July, 2005.

[76] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS'88), USA*, June, 1988.

[77] D. Liu and P. Ning. Establishing pairwise keys in distributed sensor networks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03), USA*, October, 2003.

[78] K. Liu. *Scheduling Algorithms for Instance-Intensive Cloud Workflows*. PhD Thesis, Swinburne University of Technology, Australia, 2009.

[79] 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, Special Issue on Scientific Workflows, vol. 18, no. 10, pp. 1039-1065*, 2006.

[80] M. Maheswaran, S. Ali, H.J.Siegel, D. Hensgen, and R. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computng systems. In *Proceedings of the 8th Heterogeneous Computing Workshop (HCW'99), Puerto Rico*, April, 1999.

[81] A. Mandal and et al. Scheduling strategies for mapping application workflows onto the grid. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC'05), USA*, July, 2005.

[82] N. Muscettolay, P. Nayakz, B. Pellz, and B. Williams. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence, vol. 103, no. 1-2, pp. 5-47*, 1998.

[83] G. Nadarajan. *Semantics and Planning based Workflow Composition for Video Processing*. University of Edinburgh, United Kingdom, 2010.

[84] P. Nascimento, C. Sena, J. da Silva, D. Vianna, C. Boeres, and V. Rebello. Managing the execution of large scale mpi applications on computational grids. In *Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing (SBAC-PAD'05), Brazil*, October, 2005.

[85] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics, vol. 20, no. 17, pp. 3045-3054*, 2004.

[86] S. Pandey, W. Voorsluys, M. Rahman, R. Buyya, J. Dobson, and K. Chiu. A grid workflow environment for brain imaging analysis on distributed systems. *Concurrency and Computation: Practice and Experience (CCPE), vol. 21, no. 16, pp. 2118-2139*, 2009.

[87] S. Pandey, W. Voorsluys, M. Rahman, R. Buyya, J. Dobson, and K. Chiu. Brain image registration analysis workflow for fmri study in global grids. In *Proceedings of the 23th IEEE International Conference on Advanced Information Networking and Application (AINA'09), UK*, May, 2009.

[88] S. Pandey, L. Wu, S. Guru, and R. Buyya. A particle swarm optimization (pso)-based heuristic for scheduling workflow applications in cloud computing environments. In *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications (AINA'10), Australia*, April, 2010.

[89] M. Parashar and S. Hariri. *Autonomic Computing: An Overview, Unconventional Programming Paradigms, J.-P. Banatre et al. (eds.)*. LNCS, Springer Verlag, Germanay, 2005.

[90] M. Parashar and S. Hariri. Autonomic grid computing. In *Proceedings of the 2nd International Conference on Autonomic Computing (ICAC'05), USA*, June, 2005.

[91] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. Automate: Enabling autonomic applications on the grid. *Cluster Computing: The Journal of Networks, Software Tools, and Applications, Special Issue on Autonomic Computing, vol. 9, no. 1, pp. 161-174*, 2006.

[92] J. L. Peterson. Petri nets. *ACM Computing Surveys, vol. 9, no. 3, pp. 223-252*, 1977.

[93] M. Rahman, M. R. Hassan, and R. Buyya. Jaccard based availability prediction for enterprise grids. In *Proceedings of the 10th International Conference on Computational Science (ICCS'10), The Netherlands*, May, 2010.

[94] M. Rahman, R. Ranjan, and R. Buyya. Cooperative and decentralized workflow scheduling in global grids. In *Future Generation Computer Systems, vol. 26, no. 5, pp. 753-768*. Elsevier Press, Amsterdam, The Netherlands, 2010.

[95] M. Rahman, S. Venugopal, and R. Buyya. A dynamic critical path algorithm for scheduling scientific workflow applications on global grids. In *Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing (eScience'07), India*, December, 2007.

[96] L. Ramakrishnan, M. S. C, R. Jeffrey, L. Tilson, and D. A. Reed. Grid portals for bioinformatics. In *Proceedings of the 2nd International Workshop on Grid Computing Environments (GCE), in conjunction with ACM/IEEE Conference for High Performance Computing, Networking, Storage and Analysis (SC'06), USA*, November, 2006.

[97] L. Ramakrishnan and D. Gannon. A survey of distributed workflow characteristics and resource requirements. Technical Report TR671, Indiana University, USA, September, 2008.

[98] R. Raman, M. Livny, and M. Solomon. Resource management through multilateral matchmaking. In *Proceedings of the 9th IEEE International Symposium on High-Performance Distributed Computing (HPDC00), USA*, August, 2000.

[99] R. Ranjan and R. Buyya. *Decentralized Overlay for Federation of Enterprise Clouds, Handbook of Research on Scalable Computing Technologies, K. Li et al. (eds.)*. IGI Global, USA, 2009.

[100] R. Ranjan, L. Chan, A. Harwood, S. Karunasekera, and R. Buyya. Decentralised resource discovery service for large scale federated grids. In *Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing (eScience'07), India*, December, 2007.

[101] R. Ranjan, A. Harwood, and R. Buyya. A case for cooperative and incentive-based federation of distributed clusters. *Future Generation Computer Systems, vol. 24, no. 4, pp. 280-295*, 2008.

[102] R. Ranjan, A. Harwood, and R. Buyya. Coordinated load management in peer-to-peer coupled federated grid systems. Technical Report GRIDS-TR-2008-2, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, 2008.

[103] R. Ranjan, A. Harwood, and R. Buyya. Peer-to-peer resource discovery in global grids: A tutorial. *IEEE Communications Surveys and Tutorials, vol. 10, no. 2, pp. 6-33*, 2008.

[104] R. Ranjan, M. Rahman, and R. Buyya. A decentralized and cooperative workflow scheduling algorithm. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'08), France*, May, 2008.

[105] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, USA*, August, 2001.

[106] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: A public dht service and its uses. In *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, USA*, August, 2005.

[107] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01), Germany*, November, 2001.

[108] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, USA, 2003.

[109] A. Russo and R. L. Cigno. Push/pull protocols for streaming in p2p systems. In *Proceedings of the 28th IEEE International conference on Computer Communications (INFOCOM'09) Workshops, Brazil*, April, 2009.

[110] F. Schuller and J. Qin. Towards a workflow model for meteorological simulations on the austriangrid. In *Proceedings of the 1st Austrian Grid Symposium, Austria*, December, 2005.

[111] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA-8), USA*, February, 2002.

[112] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers, vol. C-29, no. 12, pp. 1104-1113*, 1980.

[113] J. Sonnek, A. Chandra, and J. Weissman. Adaptive reputation-based scheduling on unreliable distributed infrastructures. *IEEE Transactions on Parallel and Distributed Systems, vol. 18, no. 11, pp. 1551-1564*, 2007.

[114] R. Stevens, A. Robinson, and C. Goble. mygrid: Personalised bioinformatics on the information grid. In *Proceedings of the 11th International Conference on Intelligent Systems for Molecular Biology, Australia*, July, 2003.

[115] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, USA*, August, 2001.

[116] E. Tanin, A. Harwood, and H. Samet. Using a distributed quadtree index in peer-to-peer networks. *VLDB Journal, vol. 16, no. 2, pp. 165-178*, 2007.

[117] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. *Condor - A Distributed Job Scheduler, Beowulf Cluster Computing with Linux, W. Groop et al. (eds.)*. The MIT Press, USA, 2002.

[118] I. Taylor, M. Shields, and I. Wang. *Resource Management for the Triana Peer-to-Peer Services*, pages 451–462. Grid Resource Management, J. Nabrzyski et al. (eds.). Kluwer Academic Publishers, The Netherlands, 2004.

[119] I. J. Taylor, M. S. Shields, I. Wang, and R. Philp. Distributed P2P Computing within Triana: A Galaxy Visualization Test Case. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03), France*, April, 2003.

[120] D. Theiner and P. Rutschmann. An inverse modelling approach for the estimation of hydrological model parameters. *Journal of Hydroinformatics*, 2005.

[121] H. Topcuoglu, S. Hariri, and M. Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems, vol. 13, no. 3, pp. 260-274*, 2002.

[122] H. Topcuouglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems, vol. 13, no. 3, pp. 260-274*, 2002.

[123] J. Varia. *Architecting Applications for the Amazon Cloud, Cloud Computing: Principles and Paradigms, R. Buyya et al. (eds.)*. Wiley Press, New York, USA, 2010.

[124] C. Vecchiola, X. Chu, and R. Buyya. *Aneka: A Software Platform for .NET-based Cloud Computing, High Speed and Large Scale Scientific Computing, W. Gentzsch et al. (eds.)*. IOS Press, The Netherlands, 2009.

[125] S. Venugopal and R. Buyya. A set coverage-based mapping heuristic for scheduling distributed data-intensive applications on global grids. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (Grid'06), Spain*, September, 2006.

[126] S. Venugopal, R. Buyya, and L. Winton. A grid service broker for scheduling e-science applications on global data grids. *Concurrency and Computation: Practice and Experience, vol. 18, no. 6, pp. 685-699*, 2006.

[127] L. Wang and Y. Lu. Efficient power management of heterogeneous soft real-time clusters. In *Proceedings of the 29th IEEE Real-Time Systems Symposium (RTSS'08), Spain*, December, 2008.

[128] M. Wieczorek, R. Prodan, and T. Fahringer. Scheduling of scientific workflows in the askalon grid enviornment. *ACM SIGMOD Record, vol. 34, no. 3, pp. 56-62*, 2005.

[129] X. F. Wu, V. Taylor, and R. Stevens. Design and implementation of prophesy automatic instrumentation and data entry system. In *Proceedings of the 13th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'01), USA*, August, 2001.

[130] Z. Wu, X. Liu, Z. Ni, D. Yuan, and Y. Yang. A market-oriented hierarchical scheduling strategy in cloud workflow systems. *Journal of Supercomputing*, 2011.

[131] Y. Yang, J. Chen, J. Lignier, and H. Jin. Peer-to-peer based grid workflow runtime environment of swindew-g. In *Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing (eScience'07), India*, December, 2007.

[132] J. Yao, C. K. Tham, and K. Y. Ng. Decentralized dynamic workflow scheduling for grid computing using reinforcement learning. In *Proceedings of the 14th IEEE International Conference on Networks (ICON'06), Singapore*, September, 2006.

[133] J. Yu and R. Buyya. Taxonomy of workflow management systems for grid computing. *Journal of Grid Computing, vol. 3, no. 3-4, pp. 171-200*, 2005.

[134] J. Yu and R. Buyya. *Gridbus Workflow Enactment Engine, Grid Computing: Infrastructure, Service, and Applications, L. Wang et al. (eds.)*. CRC Press, USA, 2009.

[135] J. Yu and R. Buyya. A budget constrained scheduling of workflow applications on utility grids using genetic algorithms. In *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC'06), France*, June, 2006.

[136] J. Yu and R. Buyya. A novel architecture for realizing grid workflow using tuple spaces. In *Proceedings of the 5th IEEE/ACM Workshop on Grid Computing (Grid'04), USA*, November, 2004.

[137] J. Yu, R. Buyya, and K. Ramamohanarao. *Workflow Scheduling Algorithms for Grid Computing*. Metaheuristics for Scheduling in Distributed Computing Environments, F. Xhafa and A. Abraham (eds.). Springer, Germany, 2008.

[138] J. Yu, S. Venugopal, and R. Buyya. A market-oriented grid directory service for publication and discovery of grid service providers and their services. *The Journal of Supercomputing, vol. 36, no. 1, pp. 17-31*, 2006.

[139] X. Zhang, J. L. Freschl, and J. M. Schopf. A performance study of monitoring and information services for distributed systems. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03), USA*, June, 2003.

[140] R. Zhou and K. Hwang. Powertrust: A robust and scalable reputation system for trusted peer-to-peer computing. *IEEE Transactions on Parallel and Distributed Systems, vol. 18, no. 4, pp. 460-473*, 2007.

[141] Q. Zhu and G. Agrawal. Supporting fault-tolerance for time-critical events in distributed environments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'09), USA*, November, 2009.