Utility-based Resource Management for Cluster Computing

by

Chee Shin Yeo

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

January 2008

Produced on archival quality paper

Utility-based Resource Management for Cluster Computing

Chee Shin Yeo

Supervisor: Assoc. Prof. Rajkumar Buyya

Abstract

The vision of utility computing is to offer computing services as a utility so that users only pay when they need to use. Hence, users define their service needs and expect them to be delivered by utility computing service providers. However, most current high performance computing resources which constitute clusters of computers do not consider user-centric service needs for resource management. They still adopt system-centric resource management approaches that focus on optimizing overall cluster performance.

To address this problem, we investigate how market-based resource management can enable utility-based resource management for cluster computing and have:

- developed a taxonomy to understand existing market-based resource management systems and analyze the research gap to support utility-based cluster resource management,
- designed and evaluated three resource management policies: Libra+$ to provide commodity-based pricing of cluster resources, LibraSLA to manage penalties for Service Level Agreement (SLA) based resource management, and LibraRiskD to manage the risk of deadline delay for job admission control,
- proposed the use of risk analysis by computing service providers to ensure that their essential objectives are achieved, and
- demonstrated the need for a utility computing service to adopt autonomic metered pricing.

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 advisor, Rajkumar Buyya for his intellectual guidance and continuous encouragement which ensure the successful completion of this thesis. I am totally amazed by the tremendous amount of time and effort that he contributes to work. His endless amount of energy is really an inspiration for me to do better research.

I thank sponsors of the GRIDS Laboratory, in particular, Sun Microsystems, the Australian Research Council (ARC) and Australian Department of Innovation, Industry, Science and Research (DIISR) for funding the research in this thesis. In addition, the Department and University provide the infrastructure for my research and travel support to international conferences.

I thank my advisory committee members, Ramamohanarao Kotagiri and Aaron Harwood for providing feedback on my research. I also want to express my gratitude to anonymous reviewers of my published research papers and external examiners of this thesis for spending substantial amounts of their precious time and effort to provide constructive comments that greatly improve the quality of my research.

I greatly appreciate the assistance and company from all current and previous members of the Grid Computing and Distributed Systems (GRIDS) Laboratory, especially Xingchen Chu, Chao Jin, and Krishna Nadiminti for their help on Aneka. Also, the Department's administrative staff (Pinoo Bharucha, Julien Reid, and Binh Phan) and technical staff (Thomas Weichert, David Staples, and John Horvath) always give their very best at all times.

I am very grateful to have Anthony Sulistio, Jia Yu, Elankovan Sundararajan, and Srikumar Venugopal as my fellow office mates in ICT Building Room 5.18A. We always have lots of fun and laughter together in discussions, meals, and outings.

Special thanks to my current and former housemates for their utmost support over the last five years: Zhou Zhu for sharing her joys and sorrows, Gaoqian Feng for preparing delicious meals, Tianchi Ma for his incredible speed in doing everything, Geoffrey Ayton for sharing his life experiences, and Hongwei Luo for the memories at Queen's College. And, I would not forget my good friend Xiumin Lin whom I knew from Singapore back in the army days and came over to Melbourne with me at the same time to study.

Finally, I am deeply indebted to my parents and brother for their unlimited patience and unconditional support, and thus dedicate this thesis to them.

Chee Shin Yeo
Melbourne, Australia
January 2008

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter presents a high-level overview of this thesis. It first provides the motivation to investigate utility-based resource management for cluster computing. It then identifies the principal research contributions and outlines the organization of this thesis.

1.1 Utility-based Cluster Resource Management

Our work on utility-based resource management for cluster computing is inspired by the emerging adoption of cluster computing in industry for high-performance, high-availability, and high-throughput processing. We focus on the limitations of existing cluster resource management systems in terms of supporting utility-based resource management. To solve this problem, we propose applying market-based resource management. Applying market-based resource management for utility-based cluster resource management also leads to beneficial implications on both Grid computing and commercial offering.

1.1.1 Motivation: Emerging Adoption of Cluster Computing

Next-generation scientific research involves solving Grand Challenge Applications (GCAs) that demand ever increasing amounts of computing power. Recently, a new type of High-Performance Computing (HPC) paradigm, called *cluster computing* [58][89][25], has become a more viable choice for executing these GCAs since clusters are able to offer equally high-performance with a lower price compared with traditional supercomputing systems. The emergence of clusters is initiated by a number of academic projects, such as Beowulf [104], Berkeley NOW [15], and HPVM [41] that prove the advantage of clusters over traditional HPC platforms. These advantages include low-entry costs to access supercomputing-level performance, the ability to track technologies, incrementally upgradeable system, open source development platforms, and vendor independence. As shown in Figure 1.1, clusters have been rapidly increasing their market share of the top 500 supercomputing sites over the last six years since their emergence. Today, about 80% of the top 500 supercomputing

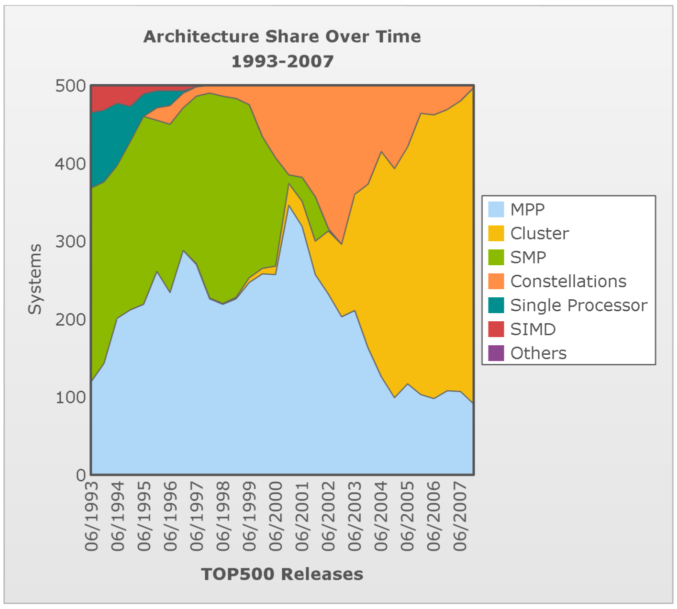


Figure 1.1: Architecture trend of top 500 supercomputing sites (1993 – 2007) (Top500 [3]).

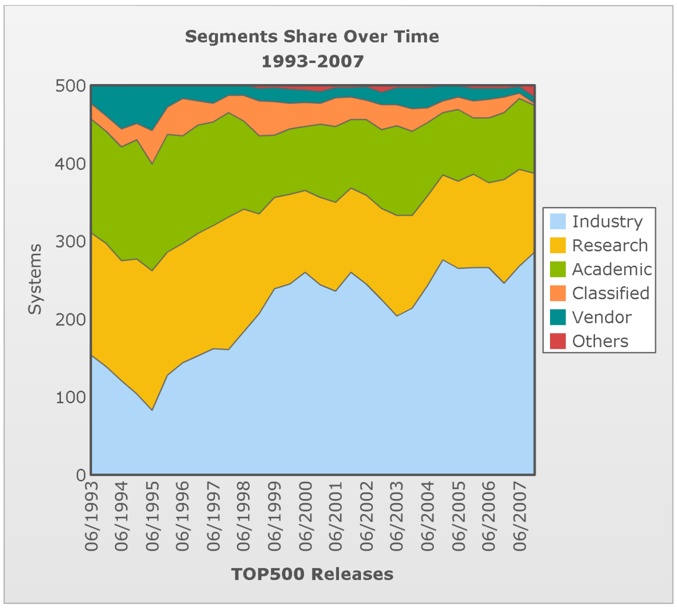


Figure 1.2: Segment trend of top 500 supercomputing sites (1993 – 2007) (Top500 [3]).

sites are clusters.

Figure 1.2 shows the segment trend of the top 500 supercomputing sites. In the past, most of the top 500 supercomputing sites are used for executing scientific research applications in academic and research institutions. However, over the last decade, an increasing number of the top 500 supercomputing sites are being used in the industry. Today, clusters are widely used for the research and development of scientific, engineering, commercial, and industrial applications that not only demand high-performance computations, but also require high availability and high throughput processing. Examples of applications using clusters include molecular dynamics simulation for protein structure prediction [111], earthquake simulation for earthquake-resistant structure design [8], petroleum reservoir simulation for oil and gas production [5], replicated storage and backup servers for satisfying huge quantity of web and database requests [21], and image rendering for fire-spread simulations to visualize destructive effects [57].

1.1.2 Problem: Limitations of Existing Cluster Resource Management Systems

A cluster is a type of parallel or distributed computer system which consists of a collection of inter-connected stand-alone computers working together as a single integrated computing resource [25]. It uses middleware to create an illusion of a single system [30] and hide the complexities of the underlying cluster architecture from the users. For example, the *cluster Resource Management System (RMS)* provides a uniform interface for user-level applications to be executed on the cluster and thus hides the existence of multiple cluster nodes from users. The cluster RMS supports four main functionalities: resource management, job queuing, job scheduling, and job execution. It manages and maintains status information of the resources such as processors and disk storage in the cluster. Jobs submitted into the cluster are initially placed into queues until there are available resources to execute the jobs. The cluster RMS then invokes a scheduler to determine how resources are assigned to jobs. After that, the cluster RMS dispatches the jobs to the assigned nodes and manages the job execution processes before returning the results to the users upon job completion.

In cluster computing, a *provider* or *producer* is the owner of the cluster that provides resources, while a *user* or *consumer* makes use of the resources provided by the cluster and can be either a physical human user or a software agent that represents a human user and acts on his behalf. Examples of resources that can be utilized in a cluster are processor power, memory storage and data storage. Thus, a single cluster can have multiple users submitting job requests that need to be completed.

Existing cluster RMSs such as Condor [115], LoadLeveler [70], Load Sharing Facility (LSF) [91], Portable Batch System (PBS) [10], and Sun Grid Engine (SGE) [108] still adopt system-centric resource allocation approaches that focus on optimizing overall cluster performance. These cluster RMSs currently maximize overall job performance and system usage of the cluster. For example, they aim to increase processor throughput and utilization for the cluster, and reduce the average waiting time and response time for the jobs. They still assume that all job requests are of equal user importance and thus neglect actual levels of service required by different users. They do not employ utility models for allocation and management of resources that would otherwise consider and thus achieve the desired utility for both cluster users and providers. Hence, these cluster RMSs neither know how users value the resources that are being competed for [27] nor provide feedback mechanisms that discourage users from submitting unlimited quantities of work [45]. Therefore, they need to be extended to support utility-based cluster computing [102]. For instance, they should support mechanisms for users to define *Quality of Service (QoS)* requirements during job submission, such as the deadline for completing a job and the budget that the user is willing to pay for completing the job before the deadline. A job may also require other essential QoS requirements in a cluster computing environment, such as the level of trust/security to safeguard the job (and its data) and the level of reliability/robustness to ensure the job does not fail to complete successfully.

1.1.3 Proposal: Market-based Resource Management

In *utility-based cluster computing*, users have different requirements and needs for various jobs and thus can assign *utility* or *value* to their job requests. Hence, for effective and efficient resource management, a cluster RMS needs to know the specific needs of different users in order to allocate resources according to their needs. During job submission to the cluster RMS, users can specify their requirements and preferences for each respective job using QoS parameters. The cluster RMS then considers these QoS parameters when making resource allocation decisions. This provides a user-centric approach with better user personalization since users can potentially affect the resource allocation outcomes based on their assigned utility. Thus, the objective of the cluster RMS is to maximize overall users' utility satisfaction. For example, the cluster RMS can achieve this objective from either the job perspective, where it maximizes the number of jobs whose QoS is satisfied or the user perspective, where it maximizes the aggregate utility perceived by individual users.

This thesis advocates the use of *market-based resource management* or *computational economy* [126][28] to achieve utility-based resource management and allocation in clusters since system-centric management for shared resources is not effective due to the lack of economic accountability. Market-based approaches can support utility-based computing within a cluster where the utility or value is the monetary payment paid by the users for accessing cluster resources. Market-based resource management is expected to regulate the supply and demand of limited cluster resources at market equilibrium, provide feedback in terms of economic incentives for both cluster users and providers, and promote QoS-based resource allocation that differentiate service requests based on their utility and thus caters to users' needs. A provider and a user have to agree on a *Service Level Agreement (SLA)* which serves as a contract outlining the expected level of service performance such that the provider is liable to compensate the user for any service under-performance. Cluster RMSs thus need to support SLA-based resource allocations that not only balance competing user needs, but also enhance the profitability of the provider while delivering the expected level of service performance. Although market-based resource management have long been proposed, there are yet any actual implemented market-based RMSs that can demonstrate they work in practice due to the lack of enabling technologies. But, with numerous recent technological advances that can aid actual deployments of market-based RMSs [103], it is now timely to examine how market-based solutions can be applied effectively even though there still remains some key challenges [103] that need to be overcome first.

This thesis builds upon an earlier work which explores the use of market-based resource management in a cluster RMS via a deadline-based proportional processor share strategy with job admission control called Libra. Libra [102] allocates a minimum proportion of processor share to each job depending on the expected remaining runtime of the job and the amount of time left till the user's specified deadline. The job admission control of Libra accepts jobs only when their deadlines can be met. However, Libra uses a static pricing function that is inflexible and not adaptive. Hence, this thesis proposes an enhanced pricing function called Libra+$ to better support utility-based resource management. Libra also does not consider the possibility of being penalized for failing to meet the required deadline of a job. Thus, this thesis describes a job admission control called LibraSLA to take into account the penalty that may be incurred for accepting new jobs. In addition, Libra requires accurate runtime estimates to ensure that the deadline of accepted jobs is met. But, runtime estimates may be rather inaccurate. Therefore, this thesis proposes a job admission control called LibraRiskD to consider the risk of deadline delay due to the inaccuracy of runtime estimates.

1.1.4 Implication: Grid Computing

The advent of *Grid computing* [54] which enables dynamic and coordinated resource sharing across various organizations, further reinforces the necessity for utility-based cluster computing. In service-oriented Grid computing [27], users can specify various levels of service required for processing their jobs on a Grid. Grid schedulers such as Grid service brokers [26][117] and Grid workflow engines [130] then make use of this user-specific information to harness resources distributed worldwide based on users' objectives. Currently, clusters dominate the majority of Grid resources whereby Grid schedulers submit and monitor their jobs being executed on the clusters through interaction with their cluster RMSs. Examples of large-scale Grids that are composed of clusters includes the TeraGrid [97] in the United States, LHC Computing Grid [78] in Europe, NAREGI [81] in Japan, and China Grid [66] in China. Grid computing is now evolving from research to industry. Commercial vendors are beginning to develop service-oriented Grid solutions for the industry since Grid computing can deliver huge business benefits to enterprises such as cost savings, easy access, and resource sharing [56]. Grid economy [29] has been proposed as a metaphor for effective management of Grid resources and application scheduling. Hence, market-based mechanisms incorporated at the cluster computing level can enforce SLAs to deliver utility and facilitate easy extensions to support Grid economy for service-oriented Grids.

1.1.5 Implication: Commercial Offering

With the advance of parallel and distributed technologies, such as cluster computing and Grid computing, commercial vendors such as Amazon [11], HP [60], IBM [62], and Sun Microsystems [109] are now progressing aggressively towards realizing a computing service market through the next era of computing model – *utility computing* or *on-demand computing* [129]. The vision of utility computing is to provide computing services whenever the users need them, thus transforming computing services to be more commoditized utilities similar to the availability of real-world utilities, such as water, electricity, gas, and telephony. With this new outsourcing service model, users no longer have to invest heavily on or maintain their own computing infrastructure, and are not constrained to specific computing service providers. Instead, they just have to pay for what they use whenever they want by outsourcing jobs to dedicated computing service providers for completion. Since users pay for using services, they want to define and expect their service needs to be delivered by computing service providers. Therefore, applying market-based mechanisms to enable utility-based cluster computing will mark a significant milestone towards realizing utility computing.

1.2 Contributions

This thesis makes the following research contributions towards the understanding and advance of utility-based resource management in cluster environments:

1. This thesis presents an abstract model to conceptualize the essential functions of a market-based cluster RMS to support utility-based cluster computing in practice. It also proposes a taxonomy consisting of five sub-taxonomies, namely the *market model*, the *resource model*, the *job model*, the *resource allocation model*, and the *evaluation model*. The taxonomy not only helps to reveal key design factors and issues that are still outstanding and crucial but also provide insights for extending and reusing components of existing market-based RMSs. Therefore, the taxonomy can lead towards more practical and enhanced market-based cluster RMSs being designed and implemented in future. This thesis then apply the taxonomy in a survey to gain a better understanding of current research progress in developing effective market-based cluster RMSs. The market-based RMSs selected for the survey are primarily research work as they reflect the latest technological advances. The design concepts and architectures of these research-based RMSs are also well-documented in publications to facilitate comprehensive comparisons, unlike commercially released products by vendors.

2. This thesis describes how the architecture of an existing cluster RMS that uses system-centric approaches can be extended to adopt market-based resource management and allocation. In addition, it presents a simple and extensible user-level job submission specification that provides a means for users to specify user-centric information such as resource and QoS requirements. It then proposes a pricing function, called Libra+$, that satisfies four essential requirements for pricing cluster resources in a commodity market model: (i) flexible, (ii) fair, (iii) dynamic, and (iv) adaptive.

3. This thesis proposes and analyzes the performance of a job admission control, called LibraSLA, that examines whether accepting a new job will affect the SLA conditions of other accepted jobs, in particular how penalties incurred on these jobs will decrease their utility. The SLA comprises four basic QoS parameters : (i) deadline type specifying whether the job can be delayed, (ii) deadline when the job needs to be finished, (iii) budget to be spent for finishing the job, and (iv) penalty rate for compensating the user for failure to meet the deadline.

4. This thesis proposes a job admission control, called LibraRiskD, that determines whether accepting a new job will expose the risk of deadline delay in the cluster. It also conducts detailed performance analysis using trace-based simulation to highlight the effectiveness of LibraRiskD in managing the risk of inaccurate runtime estimates for various scenarios that includes varying workload, deadline high:low ratio, high urgency jobs, and inaccurate runtime estimates.

5. This thesis identifies four essential objectives for a computing service provider to support the utility computing model: (i) manage wait time for SLA acceptance, (ii) meet SLA requests, (iii) ensure reliability of accepted SLA, and (iv) attain profitability. As numerous resource management policies are available, it is non-trivial to evaluate and identify the most suitable policy that truly meets the objectives of a computing service provider. Hence, this thesis presents two evaluation methods that are simple and intuitive: (i) separate and (ii) integrated risk analysis to analyze the effectiveness of resource management policies in achieving the objectives. It also provides comprehensive performance analysis of selected policies through trace-based simulation to reveal the best policy in achieving different objectives for two economic models: (i) commodity market model and (ii) bid-based model.

6. This thesis examines the need to implement an autonomic metered pricing mechanism for a utility computing service which automatically adjusts prices when necessary to maximize revenue. In particular, it highlights the significance of considering essential user requirements for autonomic metered pricing that encompass application and service requirements. Pricing computing resources according to user requirements benefits the utility computing service since different users require specific needs to be met and are willing to pay varying prices to achieve them. This thesis also describes how autonomic metered pricing can be implemented using an enterprise Grid with advanced reservations and evaluate the performance of various pricing mechanisms through experimental results to demonstrate the effectiveness of autonomic metered pricing to maximize revenue.

1.3 Organization

The rest of this thesis is organized as follows: Chapter 2 presents an abstract model and taxonomy of market-based cluster RMSs to support utility-based cluster computing in practice. Chapter 3 explains how existing cluster RMSs can be extended to support market-based mechanisms and defines an effective pricing function called Libra+$. Chapter 4 presents LibraSLA which considers the penalty incurred on jobs for delays beyond their specified deadlines. Chapter 5 describes LibraRiskD that manages the risk of deadline delay in clusters caused by inaccurate runtime estimates. Chapter 6 identifies four essential objectives for a computing service provider to support the utility computing model and proposes risk analysis methods to evaluate whether resource management policies can meet the required objectives. Chapter 7 presents the implementation of an autonomic pricing mechanism in an enterprise Grid environment using advanced reservations. Chapter 8 concludes and provides directions for future work.

The core chapters are derived from various research work that has been published during the course of candidature as detailed below:

- **Chapter 2:**
  - **C. S. Yeo** and R. Buyya. A Taxonomy of Market-based Resource Management Systems for Utility-driven Cluster Computing. *Software: Practice and Experience*, 36(13):1381-1419, 10 Nov. 2006.
- **Chapter 3:**
  - **C. S. Yeo** and R. Buyya. Pricing for Utility-driven Resource Management and Allocation in Clusters. In *Proceedings of the 12th International Conference on Advanced Computing and Communications (ADCOM 2004)*, pages 32-41, Ahmedabad, India, Dec. 2004. Allied Publishers: New Delhi, India.
  - **C. S. Yeo** and R. Buyya. Pricing for Utility-driven Resource Management and Allocation in Clusters. *International Journal of High Performance Computing Applications*, 21(4):405-418, Nov. 2007.
- **Chapter 4:**
  - **C. S. Yeo** and R. Buyya. Service Level Agreement based Allocation of Cluster Resources: Handling Penalty to Enhance Utility. In *Proceedings of the 7th IEEE International Conference on Cluster Computing (Cluster 2005)*, Boston, MA, USA, Sept. 2005. IEEE Computer Society: Los Alamitos, CA, USA.
- **Chapter 5:**
  - **C. S. Yeo** and R. Buyya. Managing Risk of Inaccurate Runtime Estimates for Deadline Constrained Job Admission Control in Clusters. In *Proceedings of the 35th International*

Conference on Parallel Processing (ICPP 2006), pages 451-458, Columbus, OH, USA, Aug. 2006. IEEE Computer Society: Los Alamitos, CA, USA.

- **Chapter 6:**
  - **C. S. Yeo** and R. Buyya. Integrated Risk Analysis for a Commercial Computing Service. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, USA, Mar. 2007. IEEE Computer Society: Los Alamitos, CA, USA.
  - **C. S. Yeo** and R. Buyya. Risk Analysis of a Commercial Computing Service in Utility Computing. *Journal of Grid Computing*. (accepted and to appear)

Chapter 2

A Taxonomy of Market-based Cluster Resource Management Systems

Recently, numerous market-based RMSs have been proposed to make use of real-world market concepts and behavior to assign resources to users for various computing platforms. This chapter presents a taxonomy that characterizes and classifies how market-based RMSs can support utility-based cluster computing in practice. The taxonomy is then mapped to existing market-based RMSs designed for both cluster and other computing platforms to survey current research developments and identify outstanding issues.

2.1 Related Work

Market-based RMSs have been utilized in many different computing platforms: clusters [43][102][63], distributed databases [14][106], Grids [6][69][76], parallel and distributed systems [80][119][24], peer-to-peer [46], and World Wide Web [9][98][77] (see Figure 2.1). They have a greater emphasis on user QoS requirements as opposed to traditional RMSs that focus on maximizing system usage.

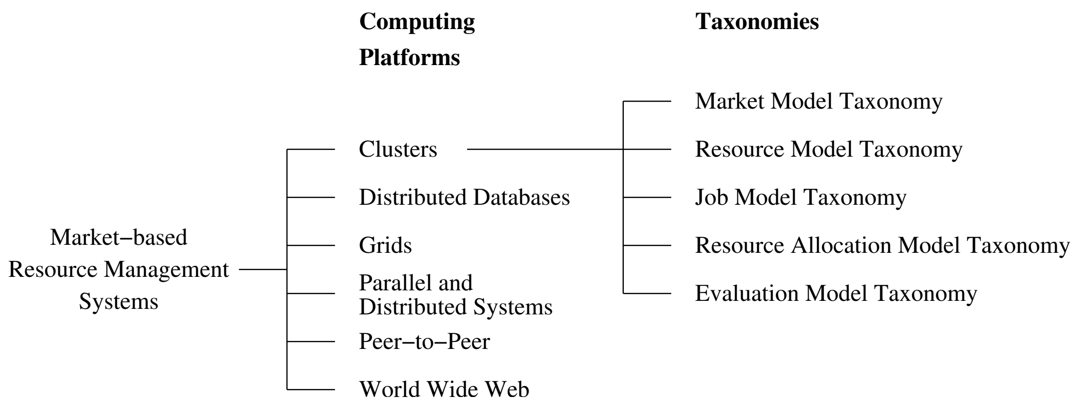


Figure 2.1: Categorization of market-based RMSs.

Market concepts can be used to prioritize competing jobs and assign resources to jobs according to users' valuations for QoS requirements and cluster resources.

Market-based cluster RMSs need to support three requirements in order to enable utility-based cluster computing [43]: (i) provide a means for users to specify their QoS needs and valuations, (ii) utilize policies to translate the valuations into resource allocations, and (iii) support mechanisms to enforce the resource allocations in order to achieve each individual user's perceived value or utility. The first requirement allows the market-based cluster RMS to be aware of user-centric service requirements so that competing service requests can be prioritized more accurately. The second requirement then determines how the cluster RMS can allocate resources appropriately and effectively to different requests by considering the solicited service requirements. The third requirement finally needs the underlying cluster operating system mechanisms to recognize and enforce the assigned resource allocations.

There are several proposed taxonomies for scheduling in distributed and heterogeneous computing. However, none of these taxonomies focus on market-based cluster computing environments. The taxonomy in [35] classifies scheduling strategies for general-purpose distributed systems. In [99], two taxonomies for state estimation and decision making are proposed to characterize dynamic scheduling for distributed systems. The EM^3 taxonomy in [49] utilizes the number of different execution modes and machine models to identify and classify heterogeneous systems. In [50], a modified version of the scheduling taxonomy in [49] is proposed to describe the resource allocation of heterogeneous systems. The taxonomy in [23] considers three characteristics of heterogeneous systems: application model, platform model, and mapping strategy to define resource matching and scheduling. A taxonomy on Grid RMS [74] includes a scheduling sub-taxonomy that examines four scheduling characteristics: scheduler organization, state estimation, rescheduling, and scheduling policy. However, our taxonomy focuses on market-based cluster RMSs for utility-based cluster computing where clusters have a number of significant differences compared with Grids. One key difference is that a cluster is distributed within a single administrative domain, whereas a Grid is distributed across multiple administrative domains.

2.2 Abstract Model for Market-based Cluster RMS

Figure 2.2 outlines an abstract model for the market-based cluster RMS. The purpose of the abstract model is to identify generic components that are fundamental and essential in a practical market-based cluster RMS and portray the interactions between these components. Thus, the abstract model can be used to explore how existing cluster RMS architectures can be leveraged and extended to incorporate market-based mechanisms to support utility-based cluster computing in practice.

The market-based cluster RMS consists of two primary entities: cluster manager and cluster

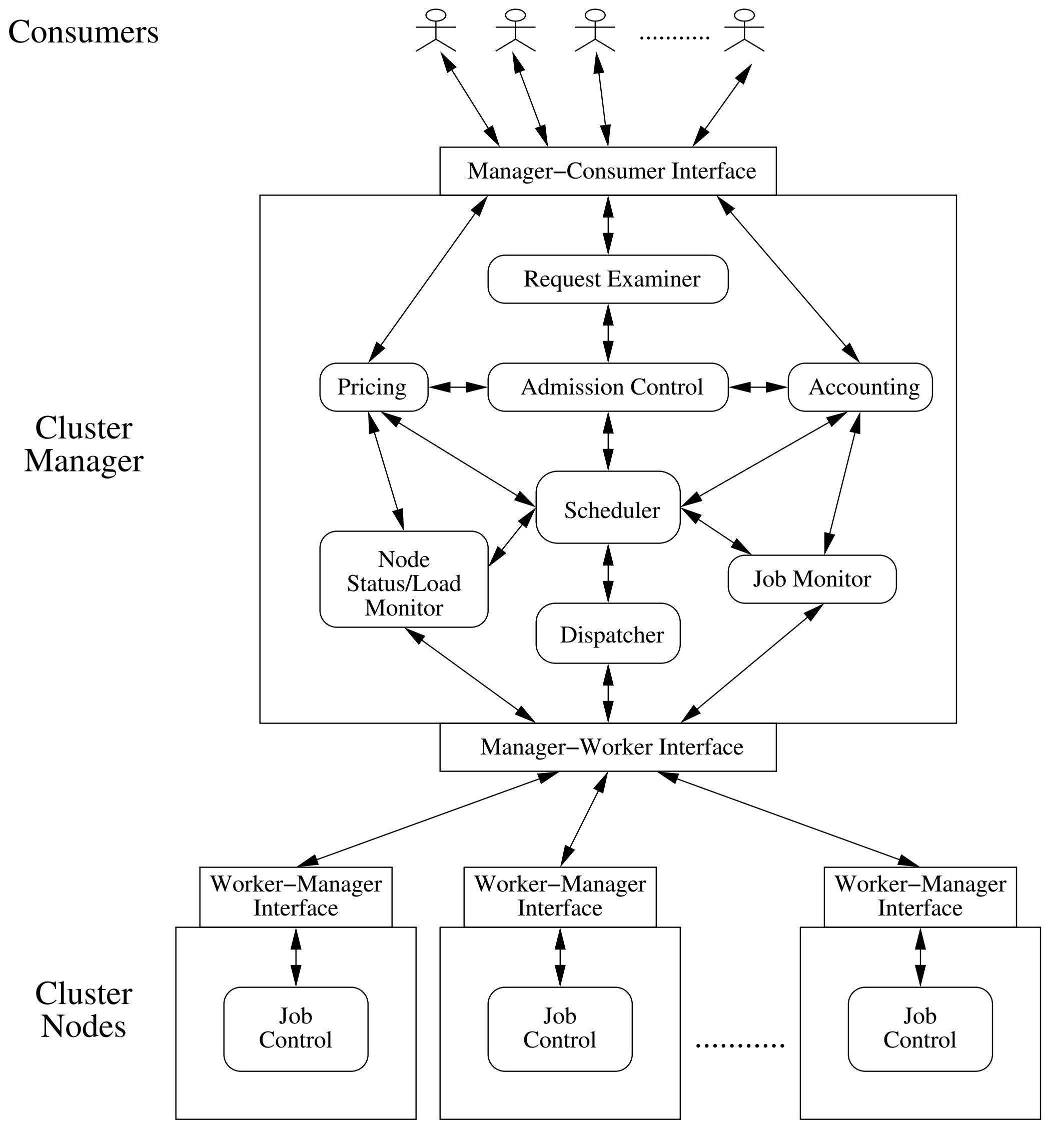


Figure 2.2: Abstract model for market-based cluster RMS.

node. For implementations within clusters, the machine that operates as the cluster manager can be known as the manager, server or master node and the machine that operates as the cluster node can be known as the worker or execution node. The actual number of cluster manager and cluster nodes depends on the implemented management control. For instance, a simple and common configuration for clusters is to support centralized management control where a single cluster manager collates multiple cluster nodes into a pool of resources as shown in Figure 2.2.

The cluster manager serves as the front-end for users and provides the scheduling engine responsible for allocating cluster resources to user applications. Thus, it supports two interfaces: the manager-consumer interface to accept requests from consumers and the manager-worker interface to execute requests on selected cluster nodes. The consumers can be actual user applications, service brokers that act on the behalf of user applications or other cluster RMSs such as those operating in multi-clustering or Grid federation environments where requests that cannot be fulfilled locally are forwarded to other cooperative clusters.

When a service request is first submitted, the request examiner interprets the submitted request for QoS requirements such as deadline and budget. The admission control then determines whether to accept or reject the request in order to ensure that the cluster is not overloaded whereby many requests cannot be fulfilled successfully. The scheduler selects suitable worker nodes to satisfy the request and the dispatcher starts the execution on the selected worker nodes. Queues can be implemented to store submitted job requests pending to be examined or accepted job requests waiting to be dispatched for execution. The scheduler and dispatcher can then be configured to select jobs from a single queue [79][84] or multiple queues [112]. The node status/load monitor keeps track of the availability of the nodes and their workload, while the job monitor maintains the execution progress of requests.

It is vital for a market-based cluster RMS to support pricing and accounting mechanisms. The pricing mechanism decides how requests are charged. For instance, requests can be charged based on submission time (peak/off-peak), pricing rates (fixed/changing) or availability of resources (supply/demand). Pricing serves as a basis for managing the supply and demand of cluster resources and facilitates in prioritizing resource allocations effectively. The accounting mechanism maintains the actual usage of resources by requests so that the final cost can be computed and charged to the consumers. In addition, the maintained historical usage information can be utilized by the scheduler to improve resource allocation decisions.

The cluster nodes provide the resources for the cluster to execute service requests via the worker-manager interface. The job control ensures that requests are fulfilled by monitoring execution progress and enforcing resource assignment for executing requests.

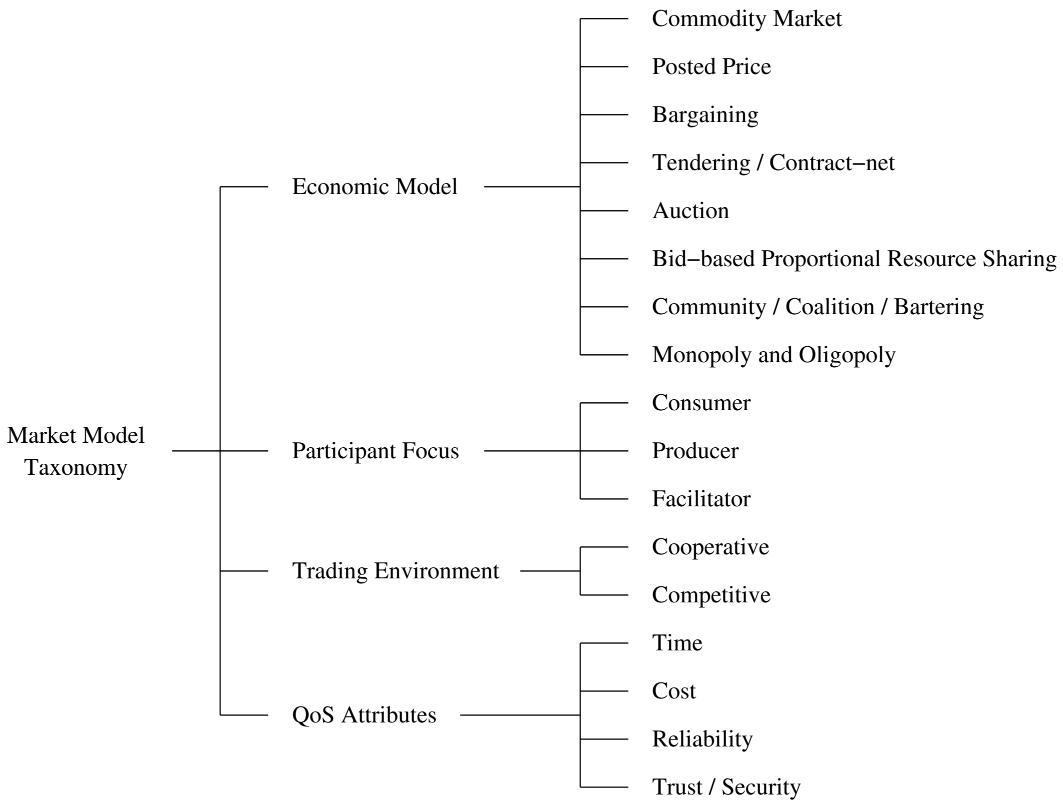


Figure 2.3: Market model taxonomy.

2.3 Taxonomy

The taxonomy emphasizes on the practical aspects of market-based cluster RMSs that are vital to achieve utility-based cluster computing in practice. It identifies key design factors and issues based on five major perspectives, namely *market model*, *resource model*, *job model*, *resource allocation model*, and *evaluation model*.

2.3.1 Market Model Taxonomy

The *market model* taxonomy examines how market concepts present in real-world human economies are incorporated into market-based cluster RMSs. This allows developers to understand what market-related attributes need to be considered, in particular to deliver utility. The market model taxonomy comprises four sub-taxonomies: *economic model*, *participant focus*, *trading environment*, and *QoS attributes* (see Figure 2.3).

Economic Model

The *economic model* derived from [28] establishes how resources are allocated in a market-driven computing environment. Selection of a suitable economic model primarily depends on the market interaction required between the consumers and producers.

For *commodity market*, producers specify prices and consumers pay for the amount of resources they consume. Pricing of resources can be determined using various parameters, such as usage time and usage quantity. There can be flat or variant pricing rates. A flat rate means that pricing is fixed for a certain time period, whereas, a variant rate means that pricing changes over time, often based on the current supply and demand at that point of time. A higher demand results in a higher variant rate.

Posted price operates similarly to the commodity market. However, special offers are advertised openly so that consumers are aware of discounted prices and can thus utilize the offers. *Bargaining* enables both producers and consumers to negotiate for a mutually agreeable price. Producers typically start with higher prices to maximize profits, but consumers start with lower prices to minimize costs. Negotiation stops when the producer or consumer does not wish to negotiate further or a mutually agreeable price has been reached. Bargaining is often used when supply and demand prices cannot be easily defined.

In *tendering/contract-net*, the consumer first announces its requirements to invite bids from potential producers. Producers then evaluate the requirements and can respond with bids if they are interested and capable of the service or ignore the announcement if they are not interested or too busy. The consumer consolidates bids from potential producers, select the most suitable producer, and send a tender to the selected producer. The tender serves as a contract and specifies conditions that the producer has to accept and conform to. Penalties may be imposed on producers if the conditions are not met. The selected producer accepts the tender and delivers the required service. The consumer then notifies other producers of the unsuccessful outcome. Tendering/contract-net allows a consumer to locate the most suitable producer to meet its service request. However, it does not always guarantee locating the best producer each time since potential producers can choose not to respond or are too busy.

Auction allows multiple consumers to negotiate with a single producer. Hence, it is different from bargaining where only one-to-one negotiation between a consumer and a producer occurs. Multiple consumers submit bids through an auctioneer who acts as the coordinator and sets the rules of the auction. Negotiation continues until a single clearing price is reached and accepted or rejected by the producer. Thus, auction regulates supply and demand based on the number of bidders, bidding price and offer price. There are basically five primary types of auctions, namely english, first-price, vickrey, dutch and double [28].

Bid-based proportional resource sharing assigns resources proportionally based on the bids provided by the consumers. Hence, each consumer is allocated a proportion of the resources as compared to a typical auction model where only one consumer with the winning bid is entitled to the resource. This is ideal for managing a large shared resource where multiple consumers are equally entitled to the resource. *Community/coalition/bartering* supports a group of community producers/consumers who shares each others' resources to create a cooperative sharing environment. This model is typically adopted in computing environments where consumers are also producers and thus both contribute and use resources. Mechanisms are required to regulate that participants act equally in both roles of producers and consumers for fairness. *Monopoly/oligopoly* depicts a non-competitive market where only a single producer (monopoly) or a number of producers (oligopoly) determines the market price. Consumers are not able to negotiate or affect the stated price from the producers.

Most market-based cluster RMSs adopt an individual economic model directly. It is also possible to use hybrids or modified variants of the economic models in order to harness the strengths of different models and provide improved customizations based on user-specific application criteria. For example, Stanford Peers Initiative [46] adopts a hybrid of auction and bartering economic models. Through an auction, the producer site selects the most beneficial consumer site with the lowest bid request for storage space, in exchange for its earlier request for storage space on the consumer site. This storage exchange between server and consumer sites creates a bartering system.

Participant Focus

The *participant focus* identifies the party for whom the market-based cluster RMS aims to achieve benefit or utility. Having a *consumer* participant focus implies that a market-based cluster RMS aims to meet the requirements specified by cluster users, and possibly optimize their perceived utility. For instance, the consumer may want to spend minimal budget for a particular job. Similarly, a *producer* participant focus results in resource owners fulfilling their desired utility. It is also possible to have a *facilitator* participant focus whereby the facilitator acts like an exchange and gains profit by coordinating and negotiating resource allocations between consumers and producers.

In utility-based cluster computing, market-based cluster RMSs need to focus primarily on achieving utility for the consumers since their key purpose is to satisfy end-users' demand for service. For example, REXEC [43] maximizes utility for consumers by allocating processing time proportionally to jobs based on their bid values. However, producers and facilitators may have specific requirements that also need to be taken into consideration and not neglected totally. For instance, Cluster-On-Demand [63] has producer participant focus as each cluster manager maximizes its earnings by accessing the risk and reward of a new job before accepting it. It is also possible for market-based RMSs to have multiple participant focus. For example, Stanford Peers Initiative [46] has both producer and consumer participant focuses as a site contributes storage space to other sites (producer), but also requests storage space in return from these sites (consumer).

Trading Environment

The *trading environment* generalizes the motive of trading between the participants that are supported via the market-based cluster RMS. The needs and aims of various participants establish the trading relationships between them. In a *cooperative* trading environment, participants work together with one another to achieve a collective benefit for them, such as producers creating a resource sharing environment that speeds up execution of jobs. On the other hand, in a *competitive* trading environment, each participant works towards its own individual benefit and does not take into account how they affect other participants, such as consumers contending with one another to secure available resources for their jobs.

A market-based cluster RMS can only support either a cooperative or competitive trading environment, but not both. For example, in Libra [102], consumers are offered incentives to encourage submitting jobs with more relaxed deadlines so that jobs from other consumers can still be accepted. REXEC [43] creates a competitive trading environment whereby consumers are allocated proportions of processing time based on their bid values; a higher bid value entitles the consumer to a larger proportion.

QoS Attributes

QoS attributes describe service requirements which consumers require the producer to provide in a service market. The *time* QoS attribute identifies the time required for various operations. Examples of time QoS attributes are job execution time, data transfer time and deadline required by the consumer for the job to be completed. The *cost* QoS attribute depicts the cost involved for satisfying the job request of the consumer. A cost QoS attribute can be monetary such as the budget that a consumer is willing to pay for the job to be completed, or non-monetary in other measurement units such as the data storage (in bytes) required for the job to be executed.

The *reliability* QoS attribute represents the level of service guarantee that is expected by the consumer. Jobs that require high reliability need the market-based cluster RMS to be highly fault-tolerant whereby check-pointing and backup facilities, with fast recovery after service failure are incorporated. The *trust/security* QoS attribute determines the level of protection needed for executing applications on resources. The trust/security QoS value of a resource can be used to reflect its types of security features supported or its trustworthiness as rated by prior users. This enables jobs requiring respective trust/security QoS values to protect their sensitive and confidential information to be allocated to the appropriate resources.

Market-based cluster RMSs need to support time, cost and reliability QoS attributes as they are critical in enabling a service market for utility-based cluster computing. The trust/security QoS attribute is also critical if the user applications require secure access to resources. For example, Libra [102] guarantees that jobs accepted into the cluster finish within the users' specified deadline (time QoS attribute) and budget (cost QoS attribute). There is no market-based cluster RMS that currently supports either reliability or trust/security QoS attribute.

Satisfying QoS attributes is highly critical in a service market as consumers pay based on the different levels of service required. The market-based cluster RMS should be able to manage service

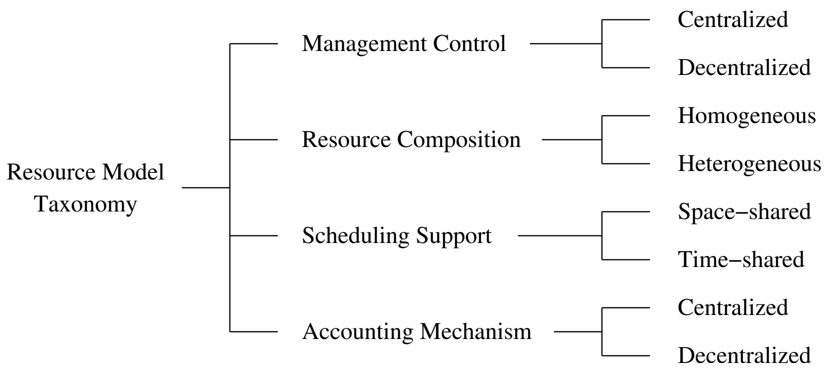


Figure 2.4: Resource model taxonomy.

demands without sacrificing existing service requests and resulting in service failures. Failure to meet QoS attributes not only requires the producer to compensate consumers, but also has a bad reputation on the producer that affects future credibility. For example, in Cluster-On-Demand [63] and LibraSLA [125], penalties are incurred for failing to satisfy the jobs' needs.

2.3.2 Resource Model Taxonomy

The *resource model* taxonomy describes architectural characteristics of clusters. It is important to design market-based cluster RMSs that conform to the clusters' underlying system architectures and operating environments since there may be certain cluster attributes that can be exploited. The resource model taxonomy comprises five sub-taxonomies: *management control*, *resource composition*, *scheduling support*, and *accounting mechanism* (see Figure 2.4).

Management Control

The *management control* depicts how the resources are managed and controlled in the clusters. A cluster with *centralized* management control has a single centralized resource manager that administers all the resources and jobs in the cluster. On the contrary, a cluster with *decentralized* management control has more than one decentralized resource managers managing subsets of resources within a cluster. Decentralized resource managers need to communicate with one another in order to be informed of local information of other managers.

A centralized resource manager collects and stores all local resource and job information within the cluster at a single location. Since a centralized resource manager has the global knowledge of the entire state of the cluster, it is easier and faster for a market-based cluster RMS to communicate and coordinate with a centralized resource manager, as opposed to several decentralized resource managers. A centralized resource manager also allows a large change to be incorporated in the cluster environment as the change needs to be updated at a single location only.

However, a centralized management control is more susceptible to bottlenecks and failures due to the overloading and malfunction of the single resource manager. A simple solution is to have backup resource managers that can be activated when the current centralized resource manager fails. In addition, centralized control architectures are less scalable compared to decentralized control architectures. Since centralized and decentralized management have various strengths and weaknesses, they perform better for different environments.

Centralized management control is mostly implemented in clusters since they are often owned by a single organization and modeled as a single unified resource. Therefore, it is sufficient for market-based cluster RMS to assume centralized management control. For instance, Libra [102] extends upon the underlying cluster RMSs such as PBS [10] which implements centralized management control.

It may be appropriate for market-based cluster RMSs to have decentralized management control for better scalability and reliability. For example, in Enhanced MOSIX [12], each cluster node makes its independent resource allocation decisions, while in REXEC [43], multiple daemons separately discover and determine the best node for a job. Decentralized management control also facilitates federation of resources across multiple clusters. For example, in a cooperative and incentive-based coupling of distributed clusters [95], every cluster has an inter-cluster coordinator in addition of the local cluster RMS to determine whether jobs should be processed locally or forwarded to other clusters.

Resource Composition

The *resource composition* defines the combination of resources that make up the cluster. A cluster with *homogeneous* resource composition consists of all worker nodes having the same resource components and configurations, whereas *heterogeneous* resource composition consists of worker nodes having different resource components and configurations.

Most clusters have homogeneous resource composition as it facilitates parallel processing of applications that require same type of resources to process. In addition, processing is also simpler and much faster since there is no need to recompile the application for different resource configurations. Therefore by default, market-based cluster RMSs assumes homogeneous resource composition.

However, it is possible that some clusters have heterogeneous resource composition since heterogeneity can allow concurrent execution of distinct applications that need different specific resources. These clusters may have different groups of worker nodes with homogeneous resource composition within each set for improved processing performance. Thus, it is ideal for market-based cluster RMSs can also support heterogeneous resource composition. These market-based cluster RMSs must also have effective means of translating requirements and measurements such as required QoS attributes and load information across heterogeneous nodes to ensure accuracy. For example, in Enhanced MOSIX [12], usages of various resources in a node are translated into a single standard cost measure to support heterogeneity.

Scheduling Support

The *scheduling support* determines the type of processing that is supported by the cluster's underlying operating system. The *space-shared* scheduling support enables only a single job to be executed at any one time on a processor, whereas the *time-shared* scheduling support allows multiple jobs to be executed at any one time on a processor. For example, most traditional cluster RMSs such as PBS [10] and SGE [108] support both space-shared and time-shared scheduling supports.

Depending on its resource allocation approach, a market-based cluster RMSs may require either space-shared or time-shared scheduling support. For space-shared scheduling support, a started job can finish earlier since it has full individual access to the processor and is thus executed faster. However, submitted jobs also need to wait longer to be started for space-shared scheduling support if no processor is free assuming that preemption is not supported.

On the other hand, time-shared scheduling support only allows shared access to processors, but may reallocate unused processing time to other jobs if a job is not using the allocated processing power such as when reading or writing data. Therefore, time-shared scheduling support can lead to a higher throughput of jobs over a period of time. Moreover, time-shared scheduling support is likely to incur less latency for executing multiple jobs as compared to space-shared scheduling support which needs to preempt the current active job and start the new one. For instance, Libra [102], REXEC [43], and Tycoon [76] utilize time-shared scheduling support to share proportions of processing power among multiple active jobs.

Accounting mechanism

The *accounting mechanism* maintains and stores information about job executions in the cluster. The stored accounting information may then be used for charging purposes or planning future resource allocation decisions. A *centralized* accounting mechanism denotes that information for the entire cluster is maintained by a single centralized accounting manager and stored on a single node. For example, REXEC [43] has a centralized accounting service to maintain credit usage for each user.

A *decentralized* accounting mechanism indicates that multiple decentralized accounting managers monitor and store separate sets of information on multiple nodes. For instance, in Tycoon [76], each local host stores accounting information to compute service cost that users need to pay and determine prices of advance resource reservation for risk-averse jobs.

Similar to the management control taxonomy, it is easier for market-based cluster RMSs to access information based on the centralized accounting mechanism. But, the centralized accounting mechanism is less reliable and scalable compared to the decentralized accounting mechanism. For more flexibility and extensibility, market-based cluster RMSs can be designed to support both centralized and decentralized accounting mechanisms. Most current implementations of market-based RMS already have built-in accounting mechanisms that maintain job execution information,

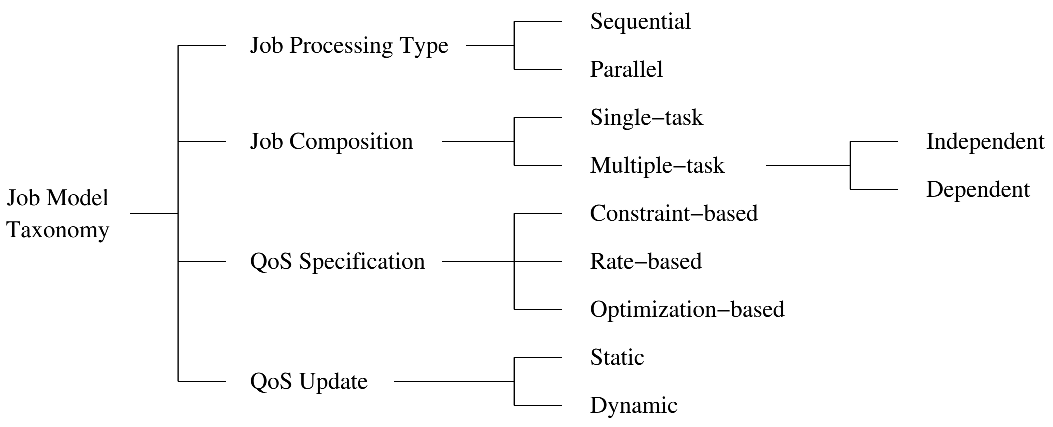


Figure 2.5: Job model taxonomy.

but may not support charging functionality which is critical to actually provide a service market.

A probable solution is to connect such market-based RMSs to specialized accounting mechanisms that support charging functionality, such as GridBank [20] and QBank [65]. In GridBank, each Grid resource uses a Grid Resource Meter to monitor the usage information and a GridBank Charging Module to compute the cost. The centralized GridBank server then transfers the payment from the users' bank accounts to the Grid resource's account. On the other hand, QBank supports both centralized and decentralized configurations. For instance, the simplest and most tightly-coupled centralized QBank configuration is having a central scheduler, bank server and database for all resources which is suitable for a cluster environment. QBank also allows multiple schedulers, bank servers and databases for each separate resource in different administrative domains to support a highly decentralized P2P or Grid environment.

2.3.3 Job Model Taxonomy

The *job model* taxonomy categorizes attributes of jobs that are to be executed on the clusters. Market-based cluster RMSs need to take into account job attributes to ensure that different job types with distinct requirements can be fulfilled successfully. The job model taxonomy comprises five sub-taxonomies: *job processing type*, *job composition*, *QoS specification*, and *QoS update* (see Figure 2.5).

Job Processing Type

The *job processing type* describes the type of processing that is required by the job. For *sequential* job processing type, the job executes on one processor independently. For *parallel* job processing type, the parallel job has to be distributed to multiple processors before executing these multiple processes simultaneously. Thus, parallel job processing type speeds up processing and is often used for solving complex problems. One common type of parallel job processing type is called message-passing where multiple processes of a parallel program on different processors interact with one another via sending and receiving messages.

All market-based cluster RMSs already support sequential job processing type which is a basic requirement for job execution. But, they also need to support parallel job processing types since most compute-intensive jobs submitted to clusters require parallel job processor type to speed up processing of complex applications. For example, Enhanced MOSIX [12] and REXEC [43] supports parallel job processing type.

Job Composition

The *job composition* portrays the collection of tasks within a job that is defined by the user. The *single-task* job composition refers to a job having a single task, while the *multiple-task* job composition refers to a job being composed of multiple tasks.

For multiple-task job composition, the tasks can be either *independent* or *dependent*. Independent tasks can be processed in parallel to minimize the overall processing time. For example, a parameter sweep job has independent multiple-task job composition since it is composed of multiple independent tasks, each with a different parameter.

On the other hand, tasks may depend on specific input data that are only available at remote locations and need to be transferred to the execution node, or are not available yet as some other jobs waiting to be processed generate the data. This means that a dependent multiple-task job composition needs to be processed in a pre-defined manner in order to ensure that all its required dependencies are satisfied. For example, a workflow job has dependent multiple-task job composition. Directed Acyclic Graphs (DAG) are commonly used to visualize the required pre-defined order of processing for workflows with no cycles, whereas workflow specification languages such as Business Process Execution Language (BPEL) [1] and XML-based workflow language (xWFL) [130] can be used to define and process workflows.

It is essential for market-based cluster RMSs to support all three job compositions: single-task, independent multiple-task, and dependent multiple-task. Single-task job composition is a basic requirement and already supported by all market-based cluster RMSs. There are also many complex scientific applications and business processes that require independent multiple-task job composition, such as parameter-sweep or dependent multiple-task job composition, such as workflow for processing. For example, Nimrod/G [6] dispatches and executes parameter-sweep jobs on a Grid. Currently, there appears to be no market-based cluster RMS that can facilitate execution of parameter-sweep or workflow jobs on clusters. Therefore, supporting all these job compositions in a market-based cluster RMS expands the community of consumers that can utilize cluster resources.

More complex mechanisms are needed to support multiple-task job composition. The market-based cluster RMS needs to schedule and monitor each task within the job to ensure that the overall job requirements can still be met successfully. For dependent multiple-task job composition, it is important to prioritize different dependencies between tasks. For example, a parent task with more dependent child tasks needs to be processed earlier to avoid delays. It is also possible to execute independent sets of dependable tasks in parallel since tasks are only dependent on one another within a set and not across sets.

QoS Specification

The *QoS specification* describes how users can specify their QoS requirements for a job to indicate their perceived level of utility. This provides cluster users with the capability to influence the resource allocation outcome in the cluster.

Users can define *constraint-based* QoS specifications that use bounded value or range of values for a particular QoS so that the minimal QoS requirements can be fulfilled. Some examples of constraint-based QoS specifications that users can specify are execution deadline, execution budget, memory storage size, disk storage size and processor power. For instance, a user can specify a deadline less than one hour (value) or deadline between one and two hours (range of values) for executing a job on cluster nodes with available memory storage size of more than 256 MB (value) and processor speed between 200 GHz and 400 GHz (range of values).

A *rate-based* QoS specification allows users to define constant or variable rates (or functions) that signify the required level of service over time. For instance, a user can specify a constant cost depreciation rate of ten credits per minute such that the user pays less for a slower job completion time. A user can also use a stochastic function to represent how the required QoS requirements will vary over time. To support a higher level of personalization, users can state *optimization-based* QoS specifications that identify specific QoS to optimize in order to maximize the users' utility. An example is a user wants to optimize the deadline of his job so that the job can be completed in the shortest possible time.

Market-based cluster RMSs need to provide any of constraint-based, rate-based or optimization-based QoS specifications so that the required utility of consumers are considered and delivered successfully. For example, REXEC [43] allows a user to specify the maximum bid value that acts as cost constraint for executing a job. Cluster-On-Demand [63] uses rate-based QoS specification where each job has a value function that depreciates at a constant rate to represent its urgency. Optimization-based QoS specification in Nimrod/G [6] minimizes time within deadline constraint, or cost within deadline constraint.

SLAs need to be negotiated and fixed between a cluster and its users to ensure that an expected level of service performance is guaranteed for submitted jobs. There are specially-designed service specification languages, such as Web Services Agreement Specification (WS-Agreement) [4] and Web Service Level Agreement (WSLA) [71] that can be utilized by market-based cluster RMSs to interpret and enforce negotiated SLAs.

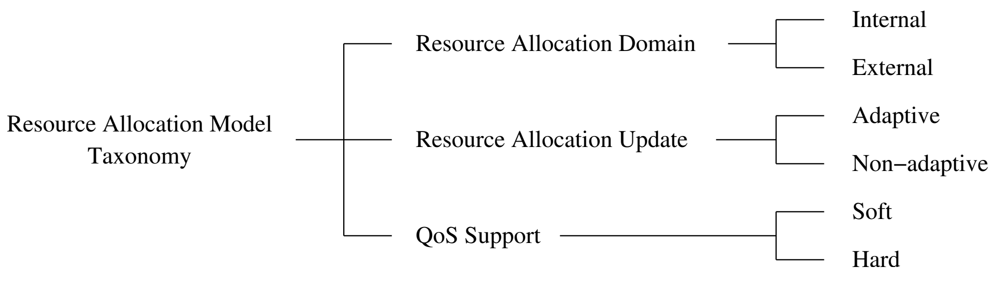

Resource Allocation Model Taxonomy
Resource Allocation Domain
Internal
External
Resource Allocation Update
Adaptive
Non–adaptive
QoS Support
Soft
Hard

Figure 2.6: Resource allocation model taxonomy.

QoS Update

The *QoS update* determines whether QoS requirements of jobs can change after jobs are submitted and accepted. The *static* QoS update means that the QoS requirements provided during job submission remain fixed and do not change after the job is submitted, while the *dynamic* QoS update means that QoS requirements of the jobs can change. These dynamic changes may already be pre-defined during job submission or modified by the user during an interactive job submission session.

Currently, all market-based cluster RMSs assumes static QoS update. For example, Libra [102] and REXEC [43] only allow users to specify QoS constraints during initial job submission. Market-based cluster RMSs also need to support dynamic QoS update so that users have the flexibility to update their latest QoS needs since it is possible that users have changing QoS needs over time. The market-based cluster RMS should also be able to reassess new changed QoS requirements and revise resource assignments accordingly as previous resource assignments may be ineffective to meet the new requirements. In addition, it is highly probable that other planned or executing jobs may also be affected so there is a need to reassess and reallocate resources to minimize any possible adverse effects.

2.3.4 Resource Allocation Model Taxonomy

The *resource allocation model* taxonomy analyzes factors that can influence how the market-based cluster RMS operates and thus affect the resource assignment outcome. The resource allocation model taxonomy comprises three sub-taxonomies: *resource allocation domain*, *resource allocation update* and *QoS support* (see Figure 2.6).

Resource Allocation Domain

The *resource allocation domain* defines the scope that the market-based cluster RMS is able to operate in. Having an *internal* resource allocation domain restricts the assignment of jobs to within the cluster. An *external* resource allocation domain allows the market-based cluster RMS to assign jobs externally outside the cluster, meaning that jobs can be executed on other remote clusters. Remote clusters may be in the same administrative domain belonging to the same producer such as an organization which owns several clusters or in different administrative domains owned by other producers such as several organizations which individually own some clusters. For instance, Cluster-On-Demand [63], Nimrod/G [6] and Stanford Peers Initiative [46] allocate jobs externally to multiple remote systems, instead of internally.

Most market-based cluster RMSs often only support internal resource allocation domains. Supporting external resource allocation domain can otherwise allow a market-based cluster RMS to have access to more alternative resources to possibly satisfy more service requests. This results in higher flexibility and scalability as service requests can still be fulfilled when there are insufficient internal resources within the clusters to meet demands. Users thus benefit since their service requests are more likely to be fulfilled. Cluster owners can also earn extra revenues by accepting external service requests. Therefore, it is ideal if market-based cluster RMSs can support external resource allocation, in addition to internal resource allocation. But, there is also the need to address other issues for supporting external resource allocation domains such as data transfer times, network delays and reliability of remote clusters.

Resource Allocation Update

The *resource allocation* update identifies whether the market-based cluster RMS is able to detect and adapt to changes to maintain effective scheduling. *Adaptive* resource allocation update means that the market-based cluster RMS is able to adjust dynamically and automatically to suit any new changes. For example, Libra [102] can allocate resources adaptively based on the actual execution and required deadline of each active job so that later arriving but more urgent jobs are allocated more resources.

On the other hand, *non-adaptive* resource allocation update means that it is not able to adapt to changes and thus still continue with its original resource assignment decision. For instance, in Stanford Peers Initiative [46], the amount of storage space allocated for data exchange remains fixed and does not change once a remote site has been selected.

In actual cluster environments, the operating condition varies over time, depending on factors such as availability of resources, amount of submission workload and users' service requirements. An initially good resource assignment decision may lead to an unfavorable outcome if it is not updated when the operating scenario changes. Likewise, there is also a possibility of improving a previously poor resource allocation decision. Therefore, market-based cluster RMSs need to support adaptive resource allocation update so that they are able to adjust and operate in changing situations to deliver a positive outcome. However, supporting adaptive resource allocation update also requires effective mechanisms to detect and determine when and how to adapt to various scenarios.

QoS Support

The *QoS support* derived from [74] determines whether QoS specified by the user can be achieved. The *soft* QoS support allows user to specify QoS parameters, but do not guarantee that these service requests can be satisfied. For example, Nimrod/G [6] provides soft QoS support as it adopts a best effort approach to schedule jobs and stop scheduling once their QoS constraints are violated.

On the contrary, the *hard* QoS support is able to ensure that the specified service can definitely be achieved. Examples of market-based cluster RMS that provide hard QoS support are Libra [102] and REXEC [43]. Libra guarantees that accepted jobs are finished within their deadline QoS, while REXEC ensures that job execution costs are limited to users' specified cost QoS.

The QoS support that a market-based cluster RMS provides should correspond to the users' service requirements. An example is the deadline QoS parameter. If a user requires hard deadline for an urgent job, it is pointless for a market-based cluster RMSs employing soft QoS support to accept the job as it does not guarantee that the deadline can be met. In reality, since different users often have various service requirements, it is best to have a market-based cluster RMSs that can provide both soft and hard QoS supports. The market-based cluster RMS can then satisfy more service requests as soft service requests can be accommodated without compromising hard QoS requests. Jobs with soft deadline may be delayed so that more jobs with hard deadline can be satisfied.

Admission control is essential during job submission to determine and feedback to the user whether the requested hard or soft QoS can be provided. If accepted by the admission control, jobs requiring hard QoS support need to be monitored to ensure that the required QoS is enforced and fulfilled. This is non-trivial as a high degree of coordination and monitoring may be necessary to enforce the QoS.

With the incorporation of penalties in SLAs, it becomes increasingly important for market-based cluster RMSs to deliver the required QoS as requested. Failure to deliver the agreed level of service can thus result in penalties that lower the financial benefits of the cluster owners. For example, penalties are modeled in Cluster-On-Demand [63] and LibraSLA [125]. In Cluster-On-Demand, jobs are penalized if they finish later than their required runtimes, whereas in LibraSLA, jobs are penalized after the lapse of their deadlines.

2.3.5 Evaluation Model Taxonomy

The *evaluation model* taxonomy outlines how to assess the effectiveness and efficiency of market-based RMSs for utility-based cluster computing. The evaluation model taxonomy comprises three sub-taxonomies: *evaluation focus*, *evaluation factors* and *overhead analysis* (see Figure 2.7).

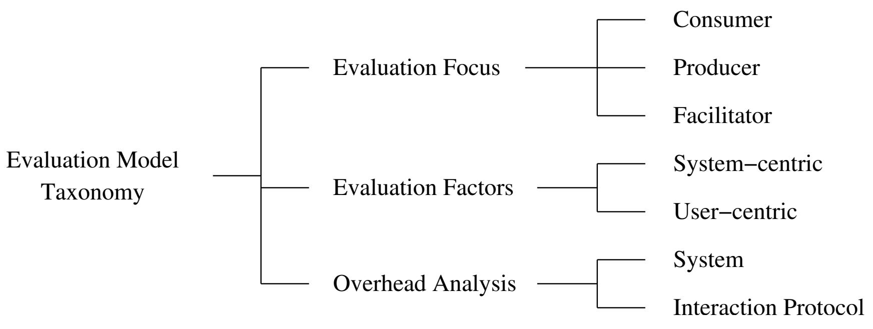


Figure 2.7: Evaluation model taxonomy.

Evaluation Focus

The *evaluation focus* identifies the party that the market-based cluster RMS is supposed to achieve utility for. The *consumer* evaluation focus measures the level of utility that has been delivered to the consumer based on its requirements. Likewise, the *producer* and *facilitator* evaluation focus evaluates how much value is gained by the producer and facilitator respectively. For example, Libra [128] evaluates the utility achieved for consumers (users) via the Job QoS Satisfaction metric and the benefits gained by the producer (cluster owner) via the Cluster Profitability metric.

The evaluation focus is similar to the participant focus sub-taxonomy discussed previously since it is logical to measure performance based on the selected participant focus. It is important to verify whether the market-based cluster RMS is able to achieve utility for the selected participant focus as expected. The evaluation focus also facilitate comparisons as market-based cluster RMSs with the same participant focus can be identified and evaluated with one another to establish similarities and differences.

Evaluation Factors

Evaluation factors are metrics defined to determine the effectiveness of different market-based cluster RMSs. *System-centric* evaluation factors measure performance from the system perspective and thus depict the overall operational performance of the cluster. Examples of system-centric evaluation factors are average waiting time, average response time, system throughput, and resource utilization. Average waiting time is the average time that a job has to wait before commencing execution, while average response time is the average time taken for a job to be completed. System throughput determines the amount of work completed in a period of time, whereas resource utilization reflects the usage level of the cluster.

User-centric evaluation factors assess performance from the participant perspective and thus portray the utility achieved by the participants. Different user-centric evaluation factors can be defined for assessing different participants that include consumer, producer or facilitator (as defined in the evaluation focus sub-taxonomy). For instance, Libra [128] defines the Job QoS Satisfaction evaluation factor for consumer evaluation focus and the Cluster Profitability evaluation factor for producer evaluation focus respectively. It is apparent that user-centric evaluation factors should constitute QoS attributes (as described in the QoS attributes sub-taxonomy) in order to assess whether the QoS required by consumers is attained. For example in Libra [128], the Job QoS Satisfaction evaluation factor computes the proportion of submitted jobs whose deadline and budget QoS parameters (time and cost in QoS attributes sub-taxonomy) are fulfilled, whereas the Cluster Profitability evaluation factor calculates the proportion of profit earned out of the total budget (cost in QoS attributes sub-taxonomy) of submitted jobs.

Both system-centric and user-centric evaluation factors are required to accurately determine the effectiveness of the market-based cluster RMS. System-centric evaluation factors ensure that system performance is not compromised entirely due to the emphasis on achieving utility, whereas user-centric evaluation factors proves that the market-based cluster RMS is able to achieve the required utility for various participants. Evaluation factors can also serve as benchmarks to grade how various market-based cluster RMSs perform for specific measures and rank them accordingly.

Overhead Analysis

The *overhead analysis* examines potential overheads that are incurred by the market-based cluster RMS. The *system* overhead analysis considers overheads sustained by the market-based cluster RMS that are of system nature. Examples of system overhead are hard disk space, memory size and processor runtime. There seems to be no market-based RMSs that explicitly mention about system overhead analysis since it is often considered to be an intrinsic system implementation issue.

The *interaction protocol* overhead analysis determines overheads that are generated by the operating policies of the market-based cluster RMS. Examples of interaction protocol overhead are communications with various nodes to determine whether they are busy or available and derive the appropriate schedule of jobs to execute on them. For instance, Tycoon [76] addresses interaction protocol overhead analysis by holding auctions internally within each service host to reduce communication across hosts.

Overheads result in system slowdowns and can create bottlenecks, thus leading to poor efficiency. There is thus a need to analyze both system and interaction protocol overheads incurred by the market-based cluster RMS in order to ensure that any overheads are kept to the minimum or within manageable limits. Otherwise, a high system overhead adds unnecessary load that burdens the cluster and reduces overall available resources to handle actual job processing. Whereas, a high interaction protocol overhead can result in longer communication time and unnecessary high network traffic that can slow down data transfers for executions. Lowering both system and interaction protocol overheads thus lead to higher scalability for handling larger number of requests which is critical for constructing viable market-based cluster RMSs.

2.4 Survey

Table 2.1 shows a summary listing of existing market-based RMSs that have been proposed by researchers for various computing platforms. Supported computing platforms include clusters, distributed databases, Grids, parallel and distributed systems, peer-to-peer, and World Wide Web. This section utilizes the taxonomy to survey some of these existing market-based RMSs (denoted by * in Table 2.1).

The market-based RMSs for the survey are chosen based on several criteria. Firstly, the survey should be concise and include sufficient number of market-based RMSs to demonstrate how the taxonomy can be applied effectively. Secondly, the selected market-based RMSs are fairly recent works so that the survey creates an insight into the latest research developments. Thirdly, the selected market-based RMSs are relevant to the scope of this thesis.

Market-based RMSs chosen for the survey can be classified into two broad categories: those proposed for clusters and other computing platforms. Since this thesis focuses on utility-based cluster computing, four market-based cluster RMSs (Cluster-On-Demand [63], Enhanced MOSIX [12], Libra [102], and REXEC [43]) are surveyed to understand current technological advances and identify outstanding issues that are yet to be explored so that more practical market-based cluster RMSs can be implemented in future.

On the other hand, surveying market-based RMSs for other computing platforms allows us to analyze and examine the applicability and suitability of these market-based RMSs for supporting utility-based cluster computing in practice. This in turn helps us to identify possible strengths of these market-based RMSs that may be leveraged for cluster computing environments. This section discusses three market-based RMSs (Faucets [69], Nimrod/G [6], and Tycoon [76]) from Grids and one market-based RMS (Stanford Peers Initiative [46]) from peer-to-peer since both Grids and peer-to-peer computing platforms are the latest and most active research areas that encompasses clusters distributed at multiple remote sites.

The survey using the various sub-taxonomies are summarized in the following tables: market model (Table 2.2), resource model (Table 2.3), job model (Table 2.4), resource allocation model (Table 2.5), and evaluation model (Table 2.6). The "NA" keyword in the tables denotes that either the specified sub-taxonomy is not addressed by the particular RMS or there is not enough information from the references to determine otherwise.

Table 2.1: Summary of market-based RMSs

Computing Platform	**Market-based RMS**	**Economic Model**	**Brief Description**
Clusters	Cluster-On-Demand * [63]	tendering/ contract-net	each cluster manager uses a heuristic to measure and balance the future risk of profit lost for accepting a job later against profit gained for accepting the job now.

Table 2.1: Continued.

Computing Platform	**Market-based RMS**	**Economic Model**	**Brief Description**
	Enhanced MOSIX * [12]	commodity market	it uses process migration to minimize the overall execution cost of machines in the cluster.
	Libra * [102]	commodity market	it offers incentives to encourage users to submit job requests with longer deadlines.
	REXEC * [43]	bid-based proportional resource sharing	it allocates resources proportionally to competing jobs based on their users' valuation.
	Utility Data Center [34]	auction	it compares two extreme auction-based resource allocation mechanisms: a globally optimal assignment market mechanism with a sub-optimal simple market mechanism.
Distributed databases	Anastasiadi et al. [14]	posted price	it examines the scenario of load balancing economy where servers advertise prices at a bulletin board and transaction requests are routed based on three different routing algorithms that focus on expected completion time and required network bandwidth.
	Mariposa [106]	tendering/ contract-net	it completes a query within its user-defined budget by contracting portions of the query to various processing sites for execution.
Grids	Bellagio [16]	auction	a centralized auctioneer computes bid values based on number of requested resources and their required durations, before clearing the auctions at fixed time periods by allocating to higher bid values first.
	CATNET [52]	bargaining	each client uses a subjective market price (computing using price quotes consolidated from available servers) to negotiate until a server quotes an acceptable price.
	Faucets * [69]	tendering/ contract-net	users specify QoS contracts for adaptive parallel jobs and Grid resources compete for jobs via bidding.
	G-commerce [121]	commodity market, auction	it compares resource allocation using either commodity market or auction strategy based on four criteria: price stability, market equilibrium, consumer efficiency, and producer efficiency.
	Gridbus [29]	commodity market	it considers the data access and transfer costs for data-oriented applications when allocating resources based on time or cost optimization.

Table 2.1: Continued.

Computing Platform	**Market-based RMS**	**Economic Model**	**Brief Description**
	Gridmarket [38]	auction	it examines resource allocation using double auction where consumers set ceiling prices and sellers set floor prices.
	Grosu and Das [59]	auction	it explores resource allocation using first-price, vickrey and double auctions.
	Maheswaran et al. [37]	auction	it investigates resource allocation based on two "co-bid" approaches that aggregate similar resources: first or no preference approaches.
	Nimrod/G * [6]	commodity market	it allocates resources to task farming applications using either time or cost optimization with deadline and budget constrained algorithms.
	OCEAN [86]	bargaining, tendering/ contract-net	it first discovers potential sellers by announcing a buyer's trade proposal and then allows the buyer to determine the best seller by using two possible negotiation mechanisms: yes/no and static bargain.
	Tycoon * [76]	auction	it allocates resources using "auction share" that estimates proportional share with consideration for latency-sensitive and risk-averse applications.
Parallel and distributed systems	Agoric Systems [82]	auction	it employs the "escalator" algorithm where users submit bids that escalate over time based on a rate and the server uses vickrey auction at fixed intervals to award resources to the highest bidder who is then charged with the second-highest bid.
	D'Agents [24]	bid-based proportional resource sharing	the server assigns resources by computing the clearing price based on the aggregate demand function of all its incoming agents.
	Dynasty [17]	commodity market	it uses a hierarchical-based brokering system where each request is distributed up the hierarchy until the accumulated brokerage cost is limited by the budget of the user.
	Enterprise [80]	tendering/ contract-net	clients broadcast a request for bids with task description and select the best bid which is the shortest estimated completion time provided by available servers.
	Ferguson et al. [53]	posted price, auction	it examines how first-price and dutch auctions can support a load balancing economy where each server host its independent auction and users decide which auction to participate based on last clearing prices advertised in bulletin boards.

Table 2.1: Continued.

Computing Platform	**Market-based RMS**	**Economic Model**	**Brief Description**
	Kurose and Simha [75]	bid-based proportional resource sharing	it uses a resource-directed approach where the current allocation of a resource is readjusted proportionally according to the marginal values computed by every agent using that resource to reflect the outstanding quantity of resource needed.
	MarketNet [124]	posted price	it advertises resource request and offer prices on a bulletin board and uses currency flow to restrict resource usage so that potential intrusion attacks into the information systems are controlled and damages caused are kept to the minimum.
	Preist et al. [93]	auction	an agent participates in mutiple auctions selling the same goods in order to secure the lowest bid possible to acquire suitable number of goods for a buyer.
	Spawn [119]	auction	it sub-divides each tree-based concurrent program into nodes (sub-programs) which then hold vickrey auction independently to obtain resources.
	Stoica et al. [105]	auction	the job with the highest bid starts execution instantly if the required number of resources are available; else it is scheduled to wait for more resources to be available and has to pay for holding on to currently available resources.
	WALRAS [120]	auction	producer and consumer agents submit their supply and demand curves respectively for a good and the equilibrium price is determined through an iterative auctioning process.
Peer-to-peer	Stanford Peers Initiative * [46]	auction, bartering	it uses data trading to create a replication network of digital archives where a winning remote site offers the lowest bid for free space on the local site in exchange for the amount of free space requested by the local site on the remote site.
World Wide Web	Java Market [13]	commodity market	it uses a cost-benefit framework to host an internet-wide computational market where producers (machines) are paid for executing consumers' jobs (Java programs) as Java applets in their web browsers.
	JaWS [77]	auction	it uses double auction to award a lease contract between a client and a host that contains the following information: agreed price, lease duration, compensation, performance statistics vector, and abort ratio.

Table 2.1: Continued.

Computing Platform	**Market-based RMS**	**Economic Model**	**Brief Description**
	POPCORN [98]	auction	each buyer (parallel programs written using POPCORN paradigm) submits a price bid and the winner is determined through one of three implemented auction mechanisms: vickrey, double, and clearinghouse double auctions.
	SuperWeb [9]	commodity market	potential hosts register with client brokers and receive payments for executing Java codes depending on the QoS provided.
	Xenoservers [96]	commodity market	it supports accounted execution of untrusted programs such as Java over the web where resources utilized by the programs are accounted and charged to the users.

Table 2.2: Survey using market model taxonomy

Market-based RMS	**Economic Model**	**Participant Focus**	**Trading Environment**	**QoS Attributes**
Cluster-On-Demand	tendering/ contract-net	producer	competitive	cost
Enhanced MOSIX	commodity market	producer	cooperative	cost
Libra	commodity market	consumer	cooperative	time, cost
REXEC	bid-based proportional resource sharing	consumer	competitive	cost
Faucets	tendering/ contract-net	producer	competitive	time, cost
Nimrod/G	commodity market	consumer	competitive	time, cost
Tycoon	auction	consumer	competitive	time, cost
Stanford Peers Initiative	auction, bartering	consumer, producer	cooperative	cost

Table 2.3: Survey using resource model taxonomy

Market-based RMS	**Management Control**	**Resource Composition**	**Scheduling Support**	**Accounting Mechanism**
Cluster-On-Demand	decentralized	NA	NA	decentralized
Enhanced MOSIX	decentralized	heterogeneous	time-shared	decentralized
Libra	centralized	heterogeneous	time-shared	centralized
REXEC	decentralized	NA	time-shared	centralized
Faucets	centralized	NA	time-shared	centralized
Nimrod/G	decentralized	heterogeneous	NA	decentralized
Tycoon	decentralized	heterogeneous	time-shared	decentralized
Stanford Peers Initiative	decentralized	NA	NA	NA

Table 2.4: Survey using job model taxonomy

Market-based RMS	**Job Processing Type**	**Job Composition**	**QoS Specification**	**QoS Update**
Cluster-On-Demand	sequential	independent single-task	rate-based	static
Enhanced MOSIX	parallel	NA	NA	NA
Libra	sequential	independent single-task	constraint-based	static
REXEC	parallel, sequential	independent single-task	constraint-based	static
Faucets	parallel	NA	constraint-based	static
Nimrod/G	sequential	independent mutiple-task	optimization-based	static
Tycoon	NA	NA	constraint-based	static
Stanford Peers Initiative	NA	NA	NA	NA

Table 2.5: Survey using resource allocation model taxonomy

Market-based RMS	**Resource Allocation Domain**	**Resource Allocation Update**	**QoS Support**
Cluster-On-Demand	external	adaptive	soft
Enhanced MOSIX	internal	adaptive	NA
Libra	internal	adaptive	hard
REXEC	internal	adaptive	hard
Faucets	internal	adaptive	soft
Nimrod/G	external	adaptive	soft
Tycoon	internal	adaptive	soft
Stanford Peers Initiative	external	non-adaptive	NA

Table 2.6: Survey using evaluation model taxonomy

Market-based RMS	**Evaluation Focus**	**Evaluation Factors**	**Overhead Analysis**
Cluster-On-Demand	producer	user-centric (cost)	NA
Enhanced MOSIX	consumer	user-centric (time)	NA
Libra	consumer, producer	system-centric, user-centric (time, cost)	NA
REXEC	consumer	user-centric (cost)	NA
Faucets	NA	NA	NA
Nimrod/G	NA	NA	NA
Tycoon	consumer	user-centric (time)	interaction protocol
Stanford Peers Initiative	consumer, producer	user-centric (reliability)	NA

2.4.1 Cluster-On-Demand

Cluster-On-Demand (COD) [36] allows the cluster manager to dynamically create independent partitions called virtual clusters (vclusters) with specific software environments for each different user groups within a cluster. This in turn facilitates external policy managers and resource brokers in the Grid to control their assigned vcluster of resources. A later work [63] examines the importance of opportunity cost in a service market where earnings for a job depreciates linearly over increasing time delay. A falling earning can become zero and instead become a penalty for not fulfilling the contract of task execution. Thus, each local cluster manager needs to determine the best job mix to balance the gains and losses for selecting a task instead of other tasks.

The task assignment among various cluster managers adopts the tendering/contract-net economic model. A user initiates an announcement bid that reflects its valuation for the task to all the cluster managers. Each cluster manager then considers the opportunity cost (gain or loss) for accepting the task and proposes a contract with an expected completion time and price. The user then selects and accepts a contract from the cluster manager which responded.

A competitive trading environment with producer participant focus is supported since each cluster manager aims to maximize its own earnings by assessing the risk and reward for bidding and scheduling a task. Earnings are paid by users to cluster managers as costs for adhering to the conditions of the contract. All cluster managers maintain information about its committed workload in order to evaluate whether to accept or reject a new task, hence exercising decentralized management control and accounting mechanism.

Tasks to be executed are assumed to single and sequential. For each task, the user provides a value function containing a constant depreciation rate to signify the importance of the task and thus the required level of service. The value function remains static after the contract has been accepted by the user. Tasks are scheduled externally to cluster managers in different administrative domains. Adaptive resource allocation update is supported as the cluster manager may delay less costly committed tasks for more costly new tasks that arrive later to minimize its losses for penalties incurred. This means that soft QoS support is provided since accepted tasks may complete later than expected.

Performance evaluation focuses on producer by using a user-centric cost evaluation factor to determine the average yield or earning each cluster manager achieves. Simulation results demonstrate that considering and balancing the potential gain of accepting a task instantly with the risk of future loss provides better returns for competing cluster managers.

2.4.2 Enhanced MOSIX

Enhanced MOSIX [12] is a modified version of MOSIX [18] cluster operating system that employs an opportunity cost approach for load balancing to minimize the overall execution cost of the cluster. The opportunity cost approach computes a single marginal cost of assigning a process to a cluster node based on the processor and memory usages of the process, thus representing a commodity market economic model. The cluster node with the minimal marginal cost is then assigned the process. This implies a cooperative trading environment with producer participant focus whereby the cost utility is measured in terms of usage level of resources.

In Enhanced MOSIX, decentralized resource control is established where each cluster node makes its independent resource assignment decisions. Heterogeneous resource composition is supported by translating usages of different resources into a single cost measure.

Enhanced MOSIX supports a time-sharing parallel execution environment where a user can execute a parallel application by first starting multiple processes on one cluster node. Each cluster node maintains accounting information about processes on its node and exchange information with other nodes periodically to determine which processes can be migrated based on the opportunity cost approach. Process migration is utilized internally within the cluster to assign or reassign processes to less loaded nodes, hence supporting adaptive resource allocation update.

Enhanced MOSIX does not address how QoS can be supported for users. For performance evaluation, it measures the slowdown of user processes, hence using a user-centric time evaluation factor. Simulation results demonstrate that using the opportunity cost approach returns a lower average slowdown of processes, thus benefiting the consumers.

2.4.3 Libra

Libra [102] is designed to be a pluggable market-based scheduler that can be integrated into existing cluster RMS architectures to support allocation of resources based on users' QoS requirements. Libra adopts the commodity market economic model that charges users using a pricing function. A later work [128] proposes an enhanced pricing function that supports four essential requirements for pricing of utility-based cluster resources: flexible, fair, dynamic, and adaptive.

The pricing function is flexible to allow easy configuration of the cluster owner to determine the level of sharing. It is also fair as resources are priced based on actual usage; jobs that use more resources are charged more. The price of resources is dynamic and is not based on a static rate. In addition, the price of resources adapts to the changing supply and demand of resources. For instance, high cluster workload results in increased pricing to discourage users from submitting infinitely and thus not overloading the cluster. This is crucial in providing QoS support since an overloaded cluster is not able to fulfill QoS requirements. In addition, incentive is offered to promote users to submit jobs with longer deadlines; a job with longer deadline is charged less compared to a job with shorter deadline.

The main objective of Libra is to maximize the number of jobs whose QoS requirements can be met, thus enabling a consumer participant focus. The enhanced pricing function [128] also improves utility for the producer (cluster owner) as only jobs with higher budgets are accepted with increasing cluster workload. Libra also considers both time and cost QoS attributes by allocating resources based on the deadline and budget QoS parameters for each job. A cooperative trading environment is implied as users are encouraged to provide a more relaxed deadline through incentives so that more jobs can be accommodated.

Libra communicates with the centralized resource manager in the underlying cluster RMS that collects information about resources in the cluster. For heterogeneous resource composition, measures such as estimated execution time are translated to their equivalent on different worker nodes. The cluster RMS needs to support time-shared execution given that Libra allocates resources to multiple executing jobs based on their required deadline. This ensures that a more urgent job with shorter remaining time to its deadline is allocated a larger processor time partition on a worker node as compared to a less urgent job. Libra uses a centralized accounting mechanism to monitor resource usage of active jobs so as to periodically reallocate the time partitions for each active job to ensure all jobs still complete within their required deadline.

Libra currently assumes that submitted jobs are sequential and single-task. Users can express two QoS constraints: deadline which the job needs to be completed and budget which the user is willing to pay. The QoS constraints cannot be updated after the job has been accepted for execution. Libra only schedules jobs to internal worker nodes within the cluster. Each worker node has a job control component that reassigns processor time partitions periodically based on the actual execution and required deadline of each active job, thus enforcing hard QoS support. This means that Libra can allocate resources adaptively to meet the deadline of later arriving but more urgent jobs.

Libra uses average waiting time and average response time as system-centric evaluation factors to evaluate overall system performance. In addition, Libra defines two user-centric evaluation factors [128]: Job QoS Satisfaction and Cluster Profitability to measure the level of utility achieved for the consumers (users) and producer (cluster owner) respectively. The Job QoS Satisfaction determines the percentage of jobs whose deadline and budget QoS is satisfied and thus examines the time and cost utility of the consumers. On the other hand, the Cluster Profitability calculates the proportion of profit obtained by the cluster owner and thus investigates the cost utility of the producer. Simulation results demonstrate that Libra performs better than traditional First-Come-First-Served scheduling approach for both system-centric and user-centric evaluation factors.

2.4.4 REXEC

REXEC [43] implements bid-based proportional resource sharing where users compete for shared resources in a cluster. It has a consumer participant focus since resources are allocated proportionally based on costs that competing users are willing to pay for a resource. Costs are defined as rates, such as credits per minute to reflect the maximum amount that a user wants to pay for using the resource.

Decentralized management control is achieved by having multiple daemons to separately discover and determine the best node to execute a job and then allowing each REXEC client to directly manage the execution of its jobs on the selected cluster nodes. The cluster nodes supports time-shared scheduling support so that multiple jobs share resources at the same time. A centralized accounting service maintains credit usage for each user in the cluster. REXEC does not consider the resource composition since it determines the proportion of resource assignment for a job purely on its user's valuation.

REXEC supports the execution of both sequential and parallel programs. Constraint-based cost limits are specified by users and remain static after job submission. The discovery and selection of nodes internal in the cluster is designed to be independent so that users have the flexibility to determine the node selection policy through their own REXEC client. Existing resource assignments are recomputed whenever a new job starts or finishes on a node, thus enabling adaptive resource allocation update. REXEC only considers a single QoS where the cost of job execution is limited to the users' specified rate. For a parallel program, the total credit required by all its processes is enforced not to exceed the cost specified by the user.

A later work [45] uses a user-centric evaluation factor: aggregate utility that adds up all the users' costs for completing jobs on the cluster. The cost charged to the user depends on the completion time of his job and decreases linearly over time until it reaches zero. Therefore, this presents a consumer evaluation focus where cost is the evaluation factor.

2.4.5 Faucets

Faucets [69] aims to provide efficient resource allocation on the computational Grid for parallel jobs by improving its usability and utilization. For better usability, users do not need to manually discover the best resources to execute their jobs or monitor the progress of executing jobs. To improve utilization, the parallel jobs are made adaptive using Charm++ [67] or adaptive MPI [22] frameworks so that they can be executed on a changing number of allocated processors during runtime on demand [68]. This allows more jobs to be executed at any one time and no processors are left unused.

Market economy is implemented to promote utilization of the computational Grid where each individual Grid resource maximizes its profit through maximum resource utilization. For each parallel job submitted, the user has to specify its QoS contract that includes requirements such as the software environment, number of processors (can be a single number, a set of numbers or range of numbers), expected completion time (and how this changes with number of processors), and the payoff that the user pays the Grid resource (and how this changes with actual job completion time). With this QoS contract, a parallel job completed by Faucets can have three possible economic outcomes: payoff at soft deadline, a decreased payoff at hard deadline (after soft deadline) and penalty after hard deadline.

Faucets uses the tendering/contract-net market economic model. First, it determines the list of Grid resources that are able to satisfy the job's execution requirements. Then, requests are sent out to each of these Grid resources to inform them about this new job. Grid resources can choose to decline or reply with a bid. The user then chooses the Grid resource when all the bids are collected.

Faucets has a producer participant focus and competitive trading environment as each Grid resource aims to maximize its own profit and resource utilization and thus compete with other resources. Faucets considers the time QoS attribute since each Grid resource that receives a new job request first checks that it can satisfy the job's QoS contract before replying with a bid. The cost QoS attribute is decided by the user who then chooses the resource to execute based on the bids of the Grid resources.

Faucets currently uses a centralized management control where the Faucets Central Server (FS) maintains the list of resources and applications that user can execute. However, the ultimate aim of Faucets is to have a distributed management control to improve scalability. Time-shared scheduling support is employed in Faucets where adaptive jobs executes simultaneously but on different proportion of allocated processors. A centralized accounting mechanism at the FS keeps track of participating Grid resources so that owners of these Grid resources can earn credits to execute jobs on other Grid resources. Faucets is primarily designed to support parallel job processing type only where the constraint-based QoS contract of a parallel job is provided at job submission and remains static throughout the execution.

In Faucets, the resource allocation domain operates in an internal manner where each Grid resource is only aware of jobs submitted via the FS and not other remote Grid resources. To maximize system utilization at each Grid resource, Faucets allocates proportional number of processors to jobs based on their QoS priorities since jobs are adaptive to changing number of processors. A new job with higher priority is allocated a larger proportion of processors, thus resulting in existing jobs entitled to shrinking proportion of processors. This results in soft QoS support. Faucets does not describe how utility-based performance can be evaluated.

2.4.6 Nimrod/G

Nimrod/G [6] is the grid-enabled version of Nimrod [7] that allows user to create and execute parameter sweep applications on the Grid. Its design is based on a commodity market economic model where each Nimrod/G broker associated with a user obtains service prices from Grid traders at each different Grid resource location. Nimrod/G supports a consumer participant focus that considers deadline (time) and budget (cost) QoS constraints specified by the user for running his application. Prices of resources thus vary between different executing applications depending on the time and selection of Grid resources that suits the QoS constraints. This means that users have to compete with one another in order to maximize their own personal benefits, thus establishing a competitive trading environment.

Each Nimrod/G broker acts on behalf of its user to discover the best resources for his application and does not communicate with other brokers, thus implementing a decentralized management control. It also has its own decentralized accounting mechanism to ensure that the multiple tasks within the parameter sweep application do not violate the overall constraints. In addition, the Nimrod/G broker is able to operate in a highly dynamic Grid environment where resources are heterogeneous since they are managed by different owners, each having their own operating policies. The broker does not need to know the scheduling support of each Grid resource as each resource feedbacks to the broker their estimated completion time for a task.

A parameter sweep application generates multiple independent tasks with different parameter values that can execute sequentially on a processor. For each parameter sweep application, the Nimrod/G broker creates a plan to assign tasks to resources that either optimizes time or cost within deadline and budget constraints or only satisfies the constraints without any optimization [31]. The QoS constraints for a parameter sweep application can only be specified before the creation of the plan and remains static when the resource broker discovers and schedules suitable resources.

The Nimrod/G broker discovers external Grid resources across multiple administrative domains. Resources are discovered and assigned progressively for the multiple tasks within an application depending on current resource availability that is beyond the control of the broker. Therefore, Nimrod/G is only able to provide soft QoS support as it tries its best to fulfill the QoS constraints. It supports some level of adaptive resource allocation update as it attempts to discover resources for remaining tasks yet to be scheduled based on the remaining budget from scheduled tasks so that the overall budget is not exceeded. It also attempts to reschedule tasks to other resources if existing scheduled tasks fail to start execution. However, Nimrod/G stops assigning remaining tasks once the deadline or budget QoS constraint is violated, thus wasting budget and time spent on already completed tasks. Nimrod/G does not describe how utility-based performance can be evaluated.

2.4.7 Tycoon

Tycoon [76] examines resource allocation in Grid environments where users are self-interested with unpredictable demands and service hosts are unreliable with changing availability. It implements a two-tier resource allocation architecture that differentiates between user strategy and allocation mechanism. The user strategy captures high-level preferences that are application-dependent and vary across users, while the allocation mechanism provides means to solicit true user valuations for more efficient execution. The separation of user strategy and allocation mechanism therefore allows both requirements not to be limited and dependent of one another.

Each service host utilizes an auction share scheduler that holds first-price auctions to determine resource allocation. The request with the highest bid is then allocated the processor time slice. The bid is computed as the pricing rate that the user pays for the required processor time, hence both time and cost QoS attributes are considered. Consumer participant focus is supported as users can indicate whether the service requests are latency-sensitive or throughput-driven. Based on these preferences, consumers have to compete with one another for Grid sites that can satisfy their service requests.

A host self-manages its local selection of applications, thus maintaining decentralized resource management. Hosts are heterogeneous since they are installed in various administrative domains and owned by different owners. Applications are assigned processor time slices so that multiple requests can be concurrently executed. Each host also keeps accounting information of its local applications to calculate the usage-based service cost to be paid by the user and determine prices of future resource reservation for risk-averse applications.

Tycoon is assumed to handle general service applications that include web and database services. Service execution requests are specified in terms of constraints such as the amount of cost he plans to spend and the deadline for completion. These constraints do not change after initial specification. Each auction share scheduler performs resource assignment internally within the service host. It also enables adaptive resource allocation update as new service requests modify and reduce the current resource entitlements of existing executing requests. This results in soft QoS support that can have a negative impact for risk-averse and latency-sensitive applications. To minimize this, Tycoon allows users to reserve resources in advance to ensure sufficient entitlements.

The performance evaluation concentrates on consumer using a user-centric time evaluation factor. A metric called scheduling error assesses whether users get their specified amount of resources and also justifies the overall fairness for all users. The mean latency is also measured for latency-sensitive applications to examine whether their requests are fulfilled. Simulation results demonstrate that the Tycoon is able to achieve high fairness and low latency compared to simple proportional-share strategy. In addition, Tycoon minimizes interaction protocol overheads by holding auctions internally within each service host to reduce communication across hosts.

2.4.8 Stanford Peers Initiative

Stanford Peers Initiative [46] employs a peer-to-peer data trading framework to create a digital archiving system. It utilizes a bid trading auction mechanism where a local site that wants to replicate its collection holds an auction to solicit bids from remote sites by first announcing its request for storage space. Each interested remote site then returns a bid that reflects the amount of disk storage space to request from the local site in return for providing the requested storage space. The local site then selects the remote site with the lowest bid for maximum benefit.

An overall cooperative trading environment with both producer and consumer participant focus is supported as a bartering system is built whereby sites exchange free storage spaces to benefit both themselves and others. Each site minimizes the cost of trading which is the amount of disk storage space it has to provide to the remote site for the requested data exchange. Stanford Peers Initiative implements decentralized management control as each site makes its own decision to select the most suitable remote sites to replicate its data collection.

Each site is external of one another and can belong to different owners. Once a remote site is selected, the specified amount of storage space remains fixed, hence implying non-adaptive resource allocation update. The job model taxonomy does not apply to Stanford Peers Initiative because the allocation of resources is expressed in terms of data exchange and not jobs.

Stanford Peers Initiative evaluates performance based on reliability against failures since the focus of a archiving system is to preserve data as long as possible. Reliability is measured using the mean time to failure (MTTF) for each local site that is both a producer and consumer. Simulation results demonstrate that sites that use bid trading achieves higher reliability than sites that trade equal amounts of space without bidding.

2.5 Gap Analysis

The gap analysis differentiates between what is required and what is currently available for market-based RMSs to support utility-based cluster computing. It first examines key attributes of utility-based cluster computing through a comparison with other computing platforms such as Grids and peer-to-peer. Then, it identifies design issues that are still outstanding for constructing practical market-based cluster RMSs to support utility-based cluster computing.

2.5.1 Characteristics of Utility-based Cluster Computing

A comparison between the characteristics of cluster computing and that of other computing platforms enables a better understanding of why market-based cluster RMSs designed for other computing platforms may not be applicable or suitable for cluster computing, and vice-versa. To be consistent with the scope of the survey, the other computing platforms for comparison is again only restricted to Grid and peer-to-peer computing.

Table 2.7: Comparison of cluster and Grid/peer-to-peer computing

Characteristic	**Cluster**	**Grid/peer-to-peer**
Administrative domain	single	multiple
Availability	committed	uncommitted
Component	homogeneous	heterogeneous
Coupling	tight	loose
Location	local	global
Ownership	single	multiple
Policy	homogeneous	heterogeneous
Size	limited	unlimited

Table 2.7 highlights how cluster computing differs from Grid and peer-to-peer computing. It can be seen that a cluster has different characteristics from a Grid or peer-to-peer system. In particular, a cluster has simpler characteristics that can be managed more easily.

A cluster is owned by a single owner and resides in a single administrative domain, while a Grid or peer-to-peer system comprises resources owned by multiple owners and distributed across multiple administrative domains. It is easier to gain access to cluster resources as a single owner can easily commit the cluster to be available for usage. On the other hand, resources in a Grid or peer-to-peer system are uncommitted and may not always be available as various owners have the authority to decline access anytime. This high level of uncertainty can thus lead to low reliability if not managed properly.

A cluster is tightly coupled, located at a single local site and limited in size, whereas a Grid or peer-to-peer system is loosely coupled with multiple remote sites distributed globally and can be unlimited in size. Hence, ensuring scalability in a Grid or peer-to-peer system is more complex and difficult than in a cluster. In a cluster, nodes typically have homogeneous components and policies to facilitate standard management and control. To the contrary, each site in a Grid or peer-to-peer system can have heterogeneous components and policies, thus requiring greater coordination effort.

In a cluster, middleware such as the cluster RMS creates a Single System Image (SSI) [30] to operate like a single computer that hides the existence of multiple cluster nodes from consumers and manages job execution from a single user interface. The cluster RMS has complete state information of all jobs and resources within the cluster. To support utility-based cluster computing, the market-based cluster RMS emphasizes on providing utility from the system perspective collectively for consumers using the cluster. It can also administer that overall resource allocation to different users is fair based on criteria such as their SLA requirements in order to maximize aggregate utility for consumers.

On the other hand, a Grid or peer-to-peer system operates like a collection of computing services. In a Grid or peer-to-peer system, each site competes with other sites whereby sites are under different ownership and have heterogeneous policies implemented by various producers (owners). In fact, each Grid or peer-to-peer site can be a cluster. Thus, the market-based RMS for each site is greedy and only maximizes utility for its individual site, without considering other sites since these sites are beyond its control. To support the heterogeneity of components and policies across sites, the market-based RMS at each site may need to communicate using multiple interfaces which is tedious. Moreover, the market-based RMS needs to consider scalability issues, such as communication and data transfer costs due to sites distributed globally over multiple administrative domains.

Therefore, market-based RMSs designed to achieve utility in Grid or peer-to-peer computing are not suitable for cluster computing since both computing platforms have distinctive characteristics and varying emphasis of utility. In addition, market-based RMSs designed for Grid or peer-to-peer systems address complexities that do not exist and are thus redundant in clusters.

2.5.2 Outstanding Issues

The survey helps to identify outstanding issues that are yet to be explored but are important for market-based cluster RMSs to support utility-based cluster computing in practice. There are currently only a few market-based RMSs designed specifically for clusters such as Cluster-On-Demand [63], Enhanced MOSIX [12], Libra [102], and REXEC [43]. Among them, only Libra and REXEC provide a consumer participant focus that is crucial for satisfying QoS-based requests to generate utility for consumers. None of these market-based cluster RMSs supports other important QoS attributes such as reliability and trust/security that are essential in a utility-based service market where consumers pay for usage and expect good quality service.

These market-based cluster RMSs mostly only support fairly simple job models with sequential job processing type and single-task job composition and static QoS update. But, more advanced job models that comprise parallel job processing type and multiple-task job composition, such as message-passing, parameter-sweep and workflow applications are typically required by users for their work execution in reality. Consumers may also need to modify their initial QoS specifications after job submission as their service requirements may vary over time, thus requiring the support of dynamic QoS update.

A larger pool of resources can be available for usage if the market-based cluster RMSs can be extended to discover and utilize external resources in other clusters or Grids. Service requests may then continue be fulfilled especially when there are insufficient internal resources within the clusters to meet demands. Current market-based cluster RMSs also do not support both soft and hard QoS supports.

For evaluation purposes, both system-centric and user-centric evaluation factors need to be defined to measure the effectiveness of market-based cluster RMSs in achieving overall system performance and actual benefits for both consumers and producers. Metrics for measuring system and interaction protocol overheads incurred by the market-based cluster RMSs are also required to evaluate their efficiency.

There is also a possibility of incorporating strengths proposed in market-based RMSs for other computing platforms in the context of cluster computing. For instance, the tendering/contract-net economic model in Faucets [69] may be applied in a cluster with decentralized management control where the consumer determines the resource selection by choosing the best node based on bids from competing cluster nodes. Optimization-based QoS specification in Nimrod/G [6] and the "auction share" scheduling algorithm in Tycoon [76] can improve utility for consumers, in particular those with latency-sensitive applications. Bartering concepts in Stanford Peers Initiative [46] can augment the level of sharing across internal and external resource allocation domains.

2.6 Summary

In this chapter, a taxonomy is proposed to characterize and categorize various market-based cluster RMSs that can support utility-based cluster computing in practice. The taxonomy emphasizes on five different perspectives: (i) market model, (ii) resource model, (iii) job model, (iv) resource allocation model, and (v) evaluation model. A survey is also conducted where the taxonomy is mapped to selected market-based RMSs designed for both cluster and other computing platforms. The survey helps to analyze the gap between what is already available in existing market-based cluster RMSs and what is still required so that outstanding research issues that can result in more practical market-based cluster RMSs being designed in future can be identified. The mapping of the taxonomy to the range of market-based RMSs has successfully demonstrated that the proposed taxonomy is sufficiently comprehensive to characterize existing and future market-based RMSs for utility-based cluster computing.

Chapter 3

Utility-based Resource Management System Architecture and Service Pricing

This chapter describes how an existing cluster RMS can be extended to support utility-based resource management using pricing based on a commodity market model. It first presents an architectural framework for a utility-based cluster RMS and a user-level job submission specification for soliciting user-centric information to make better resource allocation decisions. It then showcases the performance of a pricing function called Libra+$ that satisfies four essential requirements for pricing cluster resources using two user-centric performance evaluation metrics.

3.1 Related Work

There are a number of cluster RMSs such as Condor [115], LoadLeveler [70], Load Sharing Facility (LSF) [91], Portable Batch System (PBS) [10], and Sun Grid Engine (SGE) [108]. But, these existing Cluster RMSs adopt system-centric approaches that optimize overall cluster performance. For example, the cluster RMS aims to maximize processor throughput and utilization for the cluster, and minimize average waiting time and response time for the jobs. But, these system-centric approaches neglect and thus ignore user-centric required services that truly determine users' needs and utility. There are no or minimal means for users to define QoS requirements and their valuations during job submission so that the cluster RMS can improve the value of utility. We propose an architectural framework for extending these existing cluster RMSs to support utility-based resource management and allocation, and describes how market-based mechanisms can be incorporated to achieve that.

Maui [110] is an advanced scheduler that supports configurable job prioritization, fairness policies and scheduling policies to maximize resource utilization and minimize job response time. It provides extensive options for the administrator to configure and define various priorities of jobs to determine how resources are allocated to jobs. Maui also allows users to define QoS parameters for jobs that will then be granted additional privileges and supports advance reservation of resources where a set of resources can be reserved for specific jobs at a particular timeframe. In addition, Maui can be integrated as the scheduler for traditional cluster RMS such as Loadleveler, LSF, PBS and SGE. But, Maui does not offer economic incentives for users to submit jobs with lower priority or QoS requirements and cluster owner to share resources.

REXEC [44] is a remote execution environment for a campus-wide network of workstations, which forms part of the Berkeley Millennium Project. REXEC allows the user to specify the maximum rate (credits per minute) that he is willing to pay for processor time. The REXEC client then selects a compute node that matches the user requirements and executes the application directly on it. It uses a proportional resource allocation mechanism that allocates resources to jobs proportional to the user valuation irrespective of their job needs. However, different users may require specific QoS requirements to be fulfilled for their jobs depending on their needs. Here, we examine two common QoS requirements (deadline and budget) of equal importance. Assuming that users will only pay if their jobs are completed within these two QoS requirements, our aim is to complete more jobs with their QoS fulfilled so that more users will submit jobs to the cluster in the future due to a good reputation for service satisfaction, in turn leading to increased revenue. Hence, our market-based resource allocation mechanism allocates resources proportionally to jobs with respect to their required deadline QoS rather than their budget QoS (user valuation).

Libra [102] is an initial work done that successfully demonstrates that a market-based scheduler is able to deliver more utility to users compared to traditional scheduling policies. Libra allows users to specify QoS requirements and allocates resources to jobs proportional to their specified QoS requirements. Thus, Libra can complete more jobs with their QoS requirements satisfied as compared to system-centric scheduling policies that do not consider various QoS needs of different jobs. Currently, Libra computes a static cost that offers incentives for jobs with a more relaxed deadline so as to encourage users to submit jobs with a longer deadline. But, Libra does not consider the actual supply and demand of resources, thus users can continue to submit unlimited amount of jobs into the cluster if they have the budget. To overcome this limitation, we propose an enhanced pricing function that satisfies four essential requirements for pricing of cluster resources and prevents the cluster from overloading.

3.2 Architectural Framework

We describe an architectural framework for extending an existing system-centric cluster RMS to support utility-based resource management and allocation. Figure 3.1 shows the architectural framework for a utility-based cluster RMS. Four additional mechanisms: Pricing, Market-based Admission Control, Market-based Resource Allocation, and Job Control (shaded in Figure 3.1) are to be implemented as pluggable components into the existing cluster RMS architecture to support

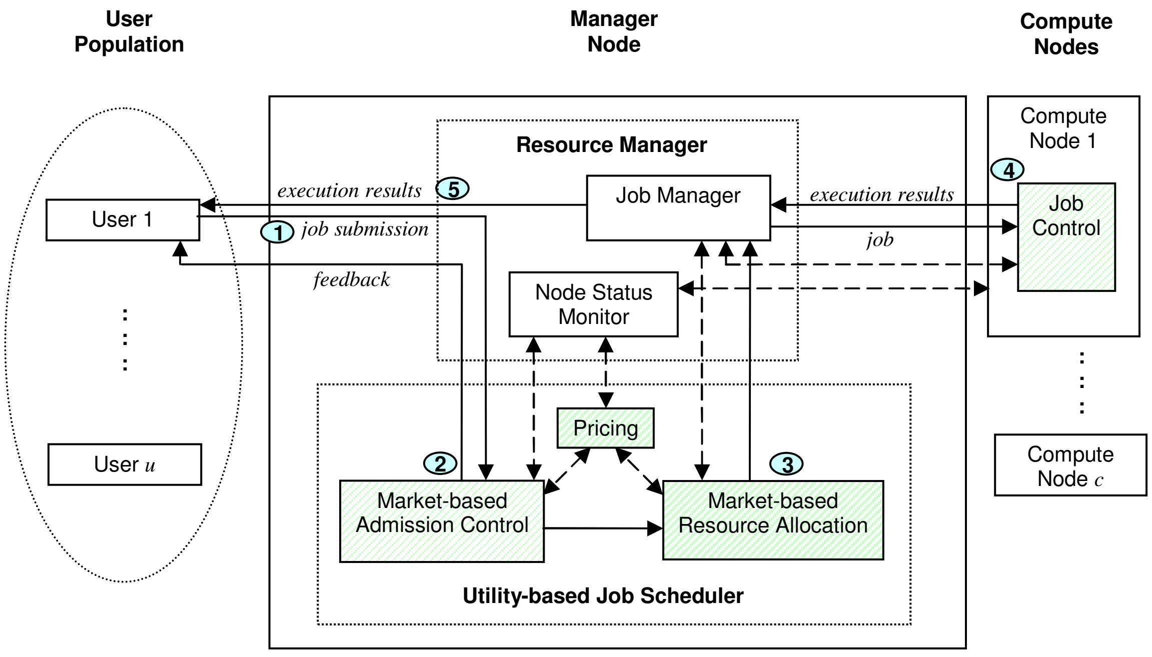


Figure 3.1: Architectural framework for a utility-based cluster RMS.

utility-based resource management.

A utility-based cluster RMS needs to determine the cost the user has to pay for executing a job and fulfilling his QoS requirements. This in turn generates economic benefits for the cluster owner to share the cluster resources. We propose a *Pricing* mechanism that employs some pricing function for this purpose. Later in this chapter, we discuss a pricing function that aims to be flexible, fair, dynamic and adaptive.

There should also be an admission control mechanism to control the number of jobs accepted into the cluster. If no admission control is implemented, an increasing number of job submissions will result in a lower number of jobs to be completed with their required QoS fulfilled due to insufficient cluster resources for too many jobs. We propose a *Market-based Admission Control* mechanism that uses dynamic and adaptive pricing (determined by the Pricing mechanism) as a natural means for admission control. For example, increasing demand of a particular resource increases its price so that fewer jobs that have sufficiently high budget will be accepted. In addition, our Market-based Admission Control mechanism also examines the required QoS of submitted jobs to admit only jobs whose QoS can be satisfied. We assume that all submitted jobs are of equal importance and only require their QoS requirements to be fulfilled. Hence, we need not first complete a job with a shorter deadline before another job with a longer deadline since the former is not regarded as more critical than the latter.

After a job is accepted, the cluster RMS needs to determine which compute node can execute the job. In addition, if there are multiple jobs waiting to be allocated, the cluster RMS needs to determine which job has the highest priority and should be allocated first. We propose a *Market-based*

Resource Allocation mechanism that considers user-centric requirements of jobs such as required resources and QoS parameters like deadline and budget, and allocate resources accordingly to these needs.

After resource allocation, there should be a mechanism to enforce the resource allocation so as to ensure that the required level of utility can be achieved. We propose a *Job Control* mechanism at each compute node that monitors and adjusts the resource allocation if necessary.

As shown in Figure 3.1, there are u local users who can submit jobs to the cluster for execution. The cluster has a single manager node and c compute nodes. The centralized resource manager of the cluster RMS is installed on the manager node to provide the user interface for users to submit jobs into the cluster. The typical flow of a job in a utility-based cluster RMS (circled numbers in Figure 3.1) is as follows:

1. A user submits a job to the cluster RMS using the user-level job submission specification.
2. The Market-based Admission Control mechanism determines whether the job should be accepted or rejected based on the job details and QoS requirements provided in the job submission specification and current workload commitments of the cluster. The outcome is feedback to the user.
3. If the job is accepted, the Market-based Resource Allocation mechanism determines which compute node the job is to be allocated to. The job manager is then informed to dispatch the job to the selected compute node.
4. The Job Control mechanism administers the execution of the job and enforces the resource allocation.
5. The job finishes execution and its execution result is returned to the user.

3.3 User-level Job Submission Specification

We propose a simple generic user-level job submission specification to capture user-centric information defined as follows:

$$job_i([Segment_1][Segment_2]...[Segment_s]) \tag{3.1}$$

Each job i submitted to the cluster has a corresponding submission specification comprising of s segments. Each segment acts as a category that contains fine-grain parameters to describe a particular aspect of job i.

The job submission specification is designed to be extensible such that new segments can be added into the specification and new parameters can be added within each segment. This flexibility can thus allow customization for gathering varying information of jobs belonging to different application models. For instance, a job belonging to a workflow-based application may have a data dependency segment.

Currently, we identify a basic job submission specification that consists of four segments for a parallel compute-intensive job:

$$\begin{aligned} job_i(\,&[JobDetails][ResourceRequirements] \\ &[QoSConstraints][QoSOptimization]) \end{aligned} \tag{3.2}$$

The first segment, $JobDetails$ describes information pertaining to the job. This provides the cluster RMS with necessary knowledge that may be utilized for more effective resource allocation. One basic example of $JobDetails$ is:

1. Runtime Estimate RE_i: The runtime estimate is the estimated time period needed to complete job i on a compute node provided that it is allocated the node's full proportion of processing power. Thus, the runtime estimate varies on nodes of different hardware and software architecture and does not include any waiting time and communication latency. The runtime estimate can also be expressed in terms of the job length in million instructions (MI).

The second segment, $ResourceRequirements$ specifies the computing resources that are needed by the job in order to be executed on a compute node. This facilitates the cluster RMS to determine whether a compute node has the necessary computing resources to execute the job. Three basic examples of $ResourceRequirements$ are:

1. Memory size MEM_i: The amount of local physical memory space needed to execute job i.

2. Disk storage size $DISK_i$: The amount of local hard disk space required to store job i.

3. Number of processors $PROC_i$: The number of processors required by job i to run. We assume that each compute node can have one or more processors. Hence, for example, if a job requires 4 processors, then the cluster RMS may allocate either 2 compute nodes with 2 processors each or 4 compute nodes with 1 processor each.

The third segment, $QoSConstraints$ states the QoS characteristics that have to be fulfilled by the cluster RMS. This captures user-centric requirements that are necessary to achieve the user's perceived utility. Two basic examples of $QoSConstraints$ are:

1. Deadline D_i: The time period in which job i has to be finished.

2. Budget B_i: The budget that the user is willing to pay for job i to be completed with the required QoS (eg. deadline) satisfied.

The fourth segment, *QoSOptimization* identifies which QoS characteristics to optimize. This supports user personalization whereby the user can determine specific QoS characteristics he wants to optimize. Two basic examples of *QoSOptimization* are:

1. Finish time TF_i: The time when job i finishes execution on a compute node. This means that the user wants the job to be finished in the shortest time, but within the specified budget.

2. Cost C_i: The actual cost the user pays to the cluster for job i provided that the required QoS is satisfied. This means that the user wants to pay the lowest cost for completing the job.

This example for a parallel compute-intensive job demonstrates the flexibility and effectiveness of the proposed generic user-level job submission specification in soliciting user-centric requirements for different application models. Users are able to express their job-specific needs and desired services that are to be fulfilled by the cluster RMS for each different job. The cluster RMS can utilize these information to determine which jobs have higher priority and allocate resources accordingly so as to maximize overall users' perceived utility, thus achieving utility-based resource management and allocation.

3.4 Libra+$: Pricing for Utility-based Resource Management

In this section, we present Libra+$ that uses a pricing function to satisfy four essential requirements for pricing of cluster resources. The admission control and resource allocation mechanisms of Libra+$ are enhanced from Libra [102] to incorporate the proposed user-level job submission specification that solicits fine-grain user-centric information for jobs and the proposed pricing function that computes dynamic and adaptive pricing for cluster resources.

3.4.1 Pricing of Cluster Resources

We outline four essential requirements for defining a pricing function to price cluster resources. First, the pricing function should be *flexible* so that it can be easily configured by the cluster owner to modify the pricing of resources to determine the level of sharing. Second, the pricing function has to be *fair*. Resources should be priced based on actual usage by the users. This means that users who use more resources pay more than users who use fewer resources. With QoS, users who specify high QoS requirements (such as a short deadline) for using a resource pay more than users who specify low QoS requirements (a long deadline). Third, the pricing function should be *dynamic* such that the price of each resource is not static and changes depending on the cluster operating condition. Fourth, the pricing function needs to be *adaptive* to changing supply and demand of resources so as to compute the relevant prices accordingly. For instance, if demand for a resource is high, the price of the resource should be increased so as to discourage users from overloading this resource and to maintain equilibrium of supply and demand of resources.

We now define a pricing function that satisfies all four essential requirements for pricing of cluster resources. Examples of cluster resources that are utilized by a job are processor time, memory size and disk storage size. The pricing function computes the pricing rate P_{ij} for per unit of cluster resource utilized by job i at compute node j as:

$$P_{ij} = (\alpha * PBase_j) + (\beta * PUtil_{ij}) \tag{3.3}$$

The pricing rate P_{ij} comprises of two components: a static component based on the base pricing rate $PBase_j$ for utilizing the resource at compute node j and a dynamic component based on the utilization pricing rate $PUtil_{ij}$ of that resource that takes into account job i. The factors α and β for the static and dynamic components respectively, provide the flexibility for the cluster owner to easily configure and modify the weight of the static and dynamic components on the overall pricing rate P_{ij}.

The cluster owner specifies the fixed base pricing rate $PBase_j$ for per unit of cluster resource. For instance, $PBase_j$ can be \$1 per second for processor time, \$2 per MB for memory size, and \$10 per GB for disk storage size. $PUtil_{ij}$ is computed as a factor of $PBase_j$ based on the utilization of the resource at compute node j from time TSU_i to TD_i, where TSU_i is the time when job i is submitted to the cluster and TD_i is the deadline time which job i has to be completed:

$$PUtil_{ij} = \frac{RESMax_j}{RESFree_{ij}} * PBase_j \tag{3.4}$$

$RESMax_j$ is the maximum units of the resource at compute node j from time TSU_i to TD_i. Assuming RES_i is the units of a resource required by job i, $RESFree_{ij}$ is the remaining free units of the resource at compute node j from time TSU_i to TD_i, after deducting units of the resource committed for other current executing jobs and job i from the maximum units of the resource:

$$RESFree_{ij} = RESMax_j - \left(\sum_{k=1}^{n_{accept_j}} RES_k \right) - RES_i \tag{3.5}$$

where $RESMax_j > RES_i$. For n jobs submitted to the cluster, n_{accept} jobs are accepted for execution by the admission control. If there is no admission control, $n_{accept} = n$. We define n_{accept_j} to be n_{accept} jobs that are executing at compute node j from time TSU_i to TD_i. To ensure $RESFree_{ij} > 0$, our Market-based Admission Control and Resource Allocation mechanisms do not allocate a job i to a compute node j if job i require units of the resource RES_i that are more than or equal to the maximum units of the resource $RESMax_j$ at compute node j.

The pricing function computes the pricing rate P_{ij} for each different resource to be used by job i at compute node j. Thus, the overall pricing rate of executing job i at compute node j can be computed as the sum of each P_{ij}. This fine-grain pricing is fair since jobs are priced based on the amount of different resources utilized. For instance, a compute-intensive job does not require a large disk storage size as compared to a data-intensive job and therefore is priced significantly lower for using the disk storage resource.

The pricing function offers incentives that take into account both user-centric and system-centric factors. The user-centric factor considered is the amount of a resource RES_i required by job i. For example, a low amount of the required resource (a low RES_i) results in a low pricing rate P_{ij}. The system-centric factor taken into account is the availability of the required resource $RESFree_{ij}$ on the compute node j. For instance, the required resource that is low in demand on node j (a high $RESFree_{ij}$) will have a low pricing rate P_{ij}.

Libra [102] offers incentives to jobs with long deadlines as compared to jobs with short deadlines irrespective of the cluster workload condition. Instead, our proposed pricing function considers the cluster workload because the utilization pricing rate $PUtil_{ij}$ considers the utilization of a resource based on the required deadline of job i (from time TSU_i to TD_i). Consider this example where the user specifies a short deadline and long deadline of 2 and 5 hours respectively to execute a job i that requires 10 units of memory size. For the compute node j, we assume that the base pricing rate $PBase_j$ is \$1 per unit, there are 100 free units of memory size for each hour of deadline, and there are n_{accept_j} jobs using 90 units of memory size during both deadlines. Hence, for a short deadline of 2 hours, $PUtil_{ij} = (200/(200-90-10)) * 1 = \$2$ per unit. Whereas, for a long deadline of 5 hours, $PUtil_{ij} = (500/(500-90-10)) * 1 = \$1.25$ per unit which is lower.

Our proposed pricing function is dynamic since the overall pricing rate of a job varies depending on the availability of each resource on different compute nodes for the required deadline. It is also adaptive as the overall pricing rate is adjusted automatically depending on the current supply and demand of resources to either encourage or discourage job submission.

3.4.2 Market-based Admission Control and Resource Allocation

We consider utility-based resource management and allocation in a simplified cluster operating environment with the following assumptions:

1. The users submit parallel compute-intensive jobs that require at least one or more processors into the cluster for execution.

2. The runtime estimate of each job is known and provided during job submission and is correct. We envision that the nature of the jobs submitted enables their runtimes to be predicted in advance based on means such as past execution history.

3. The QoS requirements specified by the user during job submission do not change after the job is accepted.

4. The cluster RMS is the only gateway for users to submit jobs to the cluster. In other words, all compute nodes in the cluster are dedicated for executing jobs that can only be assigned by the cluster RMS. This also implies that the cluster RMS has the full knowledge of allocated workload currently in execution and the resources available on each compute node.

5. The compute nodes can be homogeneous (have the same hardware architectures) or heterogeneous (have different hardware architectures). For heterogeneous compute nodes, the runtime estimate of a job is translated to its equivalent on the allocated compute node.

6. The underlying operating system at the compute nodes supports time-shared execution mechanism. A time-shared execution mechanism allows multiple jobs to be executed on a compute node at the same time. Each job shares processor time by executing within assigned processor time partitions.

Currently, our enhanced Market-based Admission Control and Resource Allocation mechanisms use a simplified version of the job submission specification in (3.2) that excludes the $QoSOptimization$ segment for the parallel compute-intensive jobs:

1. $JobDetails$:
   (a) Runtime estimate RE_i
2. $ResourceRequirements$:
   (a) Memory size MEM_i
   (b) Disk storage size $DISK_i$
   (c) Number of processors $PROC_i$
3. $QoSConstraints$:
   (a) Deadline D_i
   (b) Budget B_i

In addition, it only considers a single cluster resource which is the processor time utilized by the job. In this case, our proposed pricing function in (3.3) only computes the pricing rate for the processor time resource. Hence, $RESFree_{ij}$ which is the free processor time resource at compute node j from time TSU_i to TD_i, excluding the runtime estimates RE_k used by other current executing jobs and RE_i by job i is defined as:

$$RESFree_{ij} = RESMax_j - \left(\sum_{k=1}^{n_{accept_j}} RE_k \right) - RE_i \tag{3.6}$$

Our enhanced Market-based Admission Control and Resource Allocation mechanisms determine whether a job can be accepted or rejected based on three criteria:

1. Resources required by the job to be executed

2. Deadline that the job has to be finished

3. Budget to be paid by the user for the job to be finished within the deadline

Figure 3.2 shows the pseudocode for the enhanced Market-based Admission Control and Resource Allocation mechanisms using the proposed pricing function. First, the Admission Control mechanism determines whether there is sufficient number of compute nodes that can fulfill the specified resource requirements of job i (line 1–8). This rejects jobs that require more resources than that can be supported by the cluster. Then, the Admission Control mechanism determines whether there is a sufficient number of these compute nodes having the required resources to fulfill the required deadline time TD_i of job i (line 9–16). These compute nodes are then sorted in ascending order using $RESFree_{ij}$ in (3.6) (line 17). This ensures that each compute node is allocated jobs to its maximum capacity so that more jobs can be accepted and completed within their required deadlines. The first $PROC_i$ number of compute nodes in the sorted list that is within the specified budget B_i is then allocated to job i (line 18–34). A node j is allocated if its cost for utilizing the processor time resource based on the pricing rate P_{ij} in (3.3) is not more than the specified budget B_i of job i (line 19–25). The cost C_i of job i is thus computed as the highest cost needed from the allocated $PROC_i$ number of allocated compute nodes (line 22–24). If there is sufficient number of compute nodes meeting B_i, job i is accepted, else it is rejected (line 30–34).

3.4.3 Job Control

The Job Control mechanism at each compute node needs to enforce the QoS of a job so as to ensure that the job can finish with the required utility. Currently, we only consider enforcing a single QoS which is the deadline. We adopt the time-shared job control mechanism from Libra [102] that implements proportional-share resource allocation based on the required deadline of the job. The Job Control mechanism computes the initial processor time partition for a newly started job and then periodically updates processor time partitions for all current executing jobs to enforce that their deadlines can be satisfied.

Figure 3.3 shows the pseudocode for the Job Control mechanism that computes the processor time partition for each job i that is executing on a compute node j. The job control updates new processor time partition for every executing job i based on the processor clock time completed so far and the actual wall clock time taken with respect to its runtime estimate RE_i and deadline D_i (line 1–4).

Figure 3.2: Pseudocode for Market-based Admission Control and Resource Allocation mechanisms.

```
for $j \leftarrow 0$ to $c$ do
    if node $j$ has required resources then
        place node $j$ in $ListResReq_i$ ;
    end
end
if $ListResReq_i\_size < PROC_i$ then
    reject job $i$ with cannot_meet_resources message;
else
    for $j \leftarrow 0$ to $ListResReq_i\_size - 1$ do
        if node $j$ has required resources to finish job $i$ with $RE_i$ before $TD_i$ and $RESMax_j > RE_i$
        then
            place node $j$ in $ListNode_i$ ;
        end
    end
    if $ListNode_i\_size < PROC_i$ then
        reject job $i$ with cannot_meet_deadline message;
    else
        sort $ListNode_i$ by $RESFree_{ij}$ in ascending order;
        for $j \leftarrow 0$ to $ListNode_i\_size - 1$ do
            compute $P_{ij}$;
            if $RE_i * P_{ij} \leq B_i$ then
                place node $j$ in $ListAllocation_i$ ;
                if $ListAllocation_i\_size = 1$ or $RE_i * P_{ij} > C_i$ then
                    $C_i = RE_i * P_{ij}$;
                end
            end
            if $ListAllocation_i\_size = PROC_i$ then
                break;
            end
        end
        if $ListAllocation_i\_size < PROC_i$ then
            reject job $i$ with cannot_meet_budget message;
        else
            accept job $i$ and allocate job $i$ to all nodes in $ListAllocation_i$ ;
        end
    end
end
```

Figure 3.3: Pseudocode for Job Control mechanism.

```
for all job $i$ executing at compute node $j$ do
    get processor clock time $clockCPU_{ij}$ completed so far by node $j$ for job $i$;
    get wall clock time $clockWall_i$ currently taken by job $i$;
    set processor time partition $Partition_{ij} = \frac{RE_i - clockCPU_{ij}}{D_i - clockWall_i}$;
end
```

Table 3.1: Default simulation settings for evaluating Libra+$.

Parameter	**Default Value**
% of high urgency jobs	20.0
% of low urgency jobs	80.0
Deadline high:low ratio	4.0
Deadline low mean	2.0
Budget high:low ratio	4.0
Budget low mean	2.0
Arrival delay factor	0.5
% of inaccuracy (accurate)	0.0
Libra+$: Static pricing factor α	1.0
Libra+$: Dynamic pricing factor β	0.1

3.5 Performance Evaluation

This section first explains the evaluation methodology, before identifying the scenarios for the experiments.

3.5.1 Evaluation Methodology

We use GridSim [32] to simulate a cluster RMS environment. GridSim provides an underlying infrastructure that allows construction of simulation models for heterogeneous resources, users, applications and schedulers. GridSim has been used for the design and evaluation of market-based scheduling algorithms in both cluster [102] and Grid computing [31][33].

For the experiments, we use a subset of the last 5000 jobs in the SDSC SP2 trace (April 1998 to April 2000) version 2.2 from Feitelson's Parallel Workload Archive [2]. This 5000 job subset is based on the last 3.75 months of the SDSC SP2 trace and requires an average of 17 processors, average inter arrival time of 1969 seconds (32.8 minutes), and average runtime of 8671 seconds (2.4 hours).

As QoS parameters (deadline and budget) are not recorded in the trace, we follow a similar methodology in [63] to model these parameters through two job classes: (i) high urgency and (ii) low urgency. We model the deadline D_i and budget B_i of each job i with respect to the real runtime R_i taken from the trace. Each job in the *high urgency* class has a deadline of low D_i/R_i value and budget of high $B_i/f(R_i)$ value. $f(R_i)$ is a function to represent the minimum budget the user will quote with respect to the real runtime R_i of job i. Conversely, each job in the *low urgency* class has a deadline of high D_i/R_i value and budget of low $B_i/f(R_i)$ value. Hence, the deadline D_i and budget B_i of every job i is always assigned a higher factored value based on the real runtime R_i of the job from the trace. This model is realistic since a user who submits a more urgent job to be completed within a shorter deadline is likely to offer a higher budget for the job to be finished on time. The arrival sequence of jobs from the high urgency and low urgency classes is randomly distributed.

Values are normally distributed within each of deadline and budget parameters. The ratio of the means for each parameter's high-value and low-value is thus known as the *high:low ratio*. Hence, a higher budget high:low ratio denotes that high urgency jobs have larger budgets that of a lower ratio. For instance, a budget high:low ratio of 8 means the B_i/R_i mean of high urgency jobs is two times more than that of a budget high:low ratio of 4. Conversely, a higher deadline high:low ratio indicates that low urgency jobs have longer deadlines as compared to a lower ratio. For instance, a deadline high:low ratio of 8 signifies that the D_i/R_i mean of low urgency jobs is double than that of a deadline high:low ratio of 4.

Table 3.1 lists the default settings for our simulations. We model varying workload through the *arrival delay factor* which sets the arrival delay of jobs based on the inter arrival time from the trace. For example, an arrival delay factor of 0.1 means a job with 600 seconds of inter arrival time from the trace now has a simulated inter arrival time of 60 seconds. Hence, a lower delay factor represents higher workload by shortening the inter arrival time of jobs.

We investigate the tolerance against runtime under-estimates so as to examine the impact of delays caused by earlier jobs on later jobs, in particular failing to meet their required QoS. We do not consider over-estimated value of RE_i since jobs accepted by the admission control will still be completed within their required deadlines. The *inaccuracy* of runtime under-estimates is computed with respect to the real runtime from the trace. An inaccuracy of 0% assumes runtime estimates are equal to the real runtime of the jobs, while a higher negative percentage of inaccuracy denotes higher under-estimation.

For the operating environment, we simulate the 128-node IBM SP2 located at San Diego Supercomputer Center (SDSC) (where the selected trace originates from) with the following characteristics:

- SPEC rating of each compute node: 168
- Number of compute nodes: 128
- Processor type on each compute node: RISC System/6000
- Operating System: AIX

3.5.2 Scenarios

We evaluate the performance of our enhanced deadline-based proportional processor share policy (referred to as Libra+$) through simulation for varying workload, varying pricing factor and tolerance against runtime under-estimates. We define two user-centric performance evaluation metrics to measure the level of utility achieved by the cluster: Job QoS Satisfaction and Cluster Profitability.

Job QoS Satisfaction measures the level of utility for satisfying job requests. A higher Job QoS Satisfaction represents better performance. It is computed as the proportion of n_{QoS} jobs whose required QoS (deadline and budget) are fulfilled out of n jobs submitted:

$$\textit{Job QoS Satisfaction} = n_{QoS}/n \tag{3.7}$$

For n jobs submitted to the cluster, n_{accept} jobs are accepted for execution by the admission control of the cluster. If there is no admission control, then $n_{accept} = n$. For n_{accept} jobs accepted by the admission control, n_{QoS} jobs are completed with their required QoS satisfied. This differentiation is essential to include the possibility that a job which has been accepted by the admission control may not have its required QoS satisfied, due to inaccurate information, such as runtime underestimates resulting in delays. If all accepted jobs are completed with their required QoS satisfied, then $n_{QoS} = n_{accept}$. Currently, we only consider two basic QoS parameters of a job i: deadline D_i and budget B_i. To satisfy the deadline D_i, the finish time TF_i must be less than or equal to the deadline time TD_i. To satisfy the budget B_i, the actual cost C_i paid by the user must be less than or equal to the budget B_i.

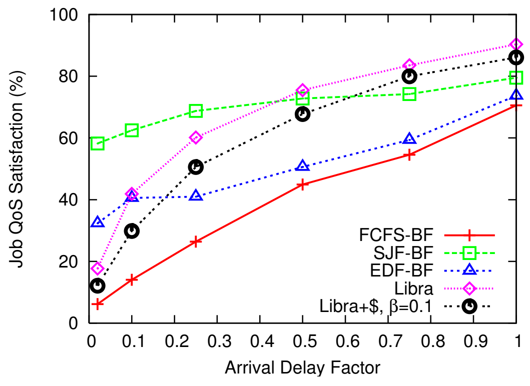
Cluster Profitability measures the level of utility for generating economic benefits for the cluster owner. A higher Cluster Profitability denotes better performance. It is computed as the proportion of profit earned by the cluster for meeting job QoS out of the total budget of jobs that are submitted:

$$\textit{Cluster Profitability} = \sum_{i=1}^{n_{QoS}} C_i / \sum_{i=1}^{n} B_i \tag{3.8}$$

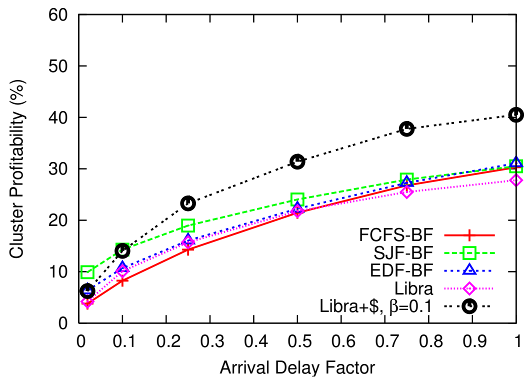
Cluster Profitability is computed as a ratio of the total budget specified by the users for submitted jobs in order to reflect the potential of the cluster in maximizing the total profit out the total budget.

To facilitate comparison, we also model three backfilling resource allocation policies: (i) FCFS-BF, (ii) SJF-BF, and (iii) EDF-BF which prioritize jobs based on arrival time (First Come First Serve), runtime estimate (Shortest Job First), and deadline (Earliest Deadline First) respectively, in addition to Libra [102] and Libra+\$. These three policies are implemented as variations of EASY backfilling [79][84] that assigns unused processors to the next waiting jobs in a single queue based on their runtime estimates, provided that they do not delay the first job with highest priority. This means that each job needs to wait for its required number of processors to be available in order to run since only a single job can run on a processor at any time (i.e. space-shared).

We select EASY backfilling for comparison since it is currently supported and widely used in a number of schedulers, including LoadLeveler [70], LSF [91], and Maui [110]. However, these backfilling policies have very poor performance without job admission control as they are not able to restrict the number of jobs which can overload the cluster. Hence, we have modified these policies not to run any waiting jobs that have exceeded their deadlines from the queue; the removed jobs



(a) Job QoS Satisfaction



(b) Cluster Profitability

Figure 3.4: Impact of decreasing workload.

are simply treated as rejected. This generous admission control not only allows the policies to choose their highest priority job at the latest time, but also ensures that jobs whose deadlines have lapsed are not unnecessarily run to incur propagated delay for later jobs.

In order to measure the Cluster Profitability metric, we also model the three backfilling policies to incorporate a simple pricing mechanism. The profit of processing a job is only considered when the deadline of the job is met. The user is then charged based on the static base pricing rate $PBase_j$ of using processing time on node j, so job i has its cost $C_i = RE_i * PBase_j$.

Libra [102] uses a pricing function that offers incentives for jobs with more relaxed deadlines to compute a static cost, so job i has its cost $C_i = \gamma * RE_i + \delta * RE_i/D_i$. γ is a factor for the first component that computes the cost based on the runtime of the job, so that longer jobs are charged more than shorter jobs. δ is a factor for the second component that offers incentives for jobs with more relaxed deadlines, so as to encourage users to submit jobs with longer deadlines. For our experiments, $\gamma = 1$ and $\delta = 1$. Libra is used to evaluate the effectiveness of the proposed pricing function in Libra+$ for improving utility for the cluster owner.

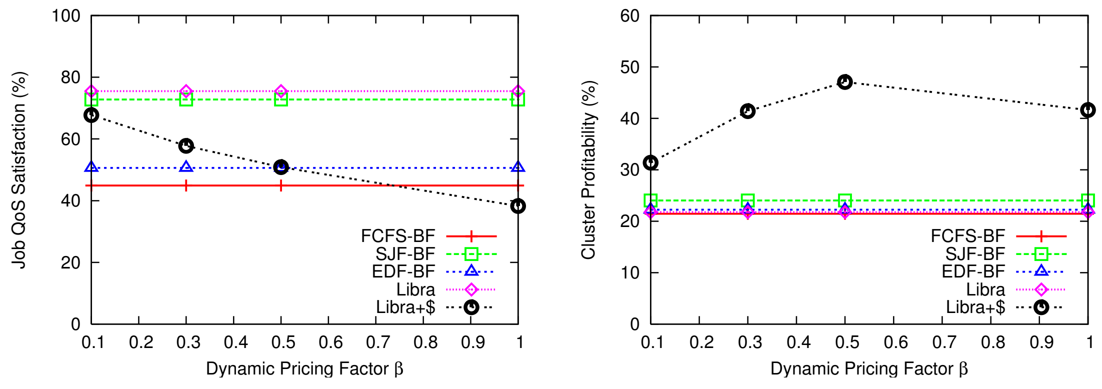
3.6 Performance Results

In this section, we examine the performance of Libra+$ for the following three scenarios: (i) varying workload, (ii) varying pricing factor, and (iii) tolerance against runtime under-estimates.

3.6.1 Varying Workload

Figure 3.4 shows how Libra+$ performs with respect to other policies for changing cluster workload. We can see that a decreasing workload (increasing arrival delay factor) results in an increasing Job QoS Satisfaction and Cluster Profitability as more jobs can have their deadlines met.

For Job QoS Satisfaction (Figure 3.4(a)), both Libra and Libra+$ are able to achieve substantially higher performance than FCFS-BF and EDF-BF since they consider the required QoS



(a) Job QoS Satisfaction (b) Cluster Profitability

Figure 3.5: Impact of increasing dynamic pricing factor β.

(deadline and budget) of each different job and allocate resources proportionally to each job based on its required deadline so that more jobs can be satisfied. However, Libra+$ has a lower Job QoS satisfaction as compared to Libra. This is because the proposed pricing function computes higher pricing, thus denying jobs with insufficient budgets.

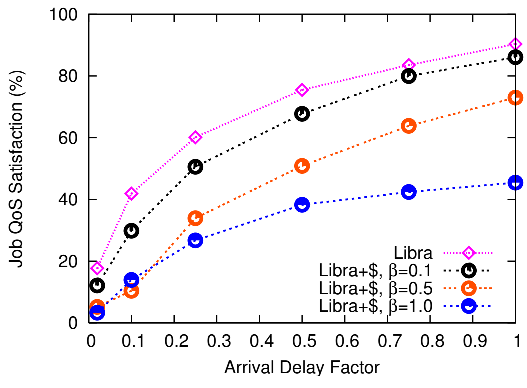
When the workload is high (with lower arrival delay factor), both Libra and Libra+$ have a lower Job QoS Satisfaction than SJF-BF and EDF-BF. This is due to the fact that in our experiments, the deadlines are always set as larger factors of runtime. Thus, SJF-BF and EDF-BF are able to exploit this as they have a more favorable selection choice due to their generous admission controls that we implemented to reject jobs only when their deadlines have lapsed, and not when they are submitted. However, both Libra and Libra+$ are able to overcome this unfairness, as seen in achieving higher Job QoS Satisfaction than SJF-BF and EDF-BF as workload decreases.

On the contrary, Figure 3.4(b) shows that the proposed pricing function enables Libra+$ to continue generating significantly higher Cluster Profitability than Libra and SJF-BF even though fewer jobs are accepted, thus proving its effectiveness in improving the cluster owner's economic benefits. Accepting fewer but higher-priced jobs enables Libra+$ to maintain a higher Cluster Profitability to compensate for a lower Job QoS Satisfaction.

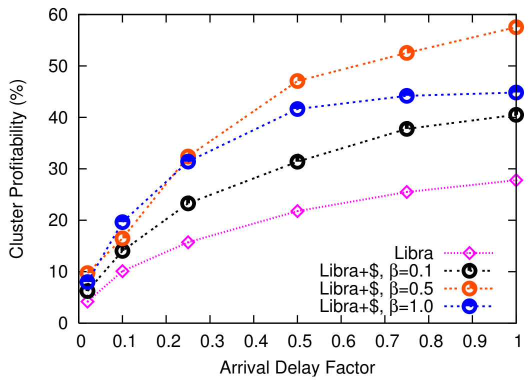
Another interesting point to note is that EDF-BF returns a similar Cluster Profitability as FCFS-BF even though it has a higher Job QoS Satisfaction. This may be due to more urgent jobs being completed first in EDF-BF, but in turn resulting in other less urgent jobs exceeding their deadlines, thus returning similar Cluster Profitability as FCFS-BF.

3.6.2 Varying Pricing Factor

We increase the dynamic pricing factor β to observe its impact on Libra+$ in supporting the level of sharing. Figure 3.5(a) shows that an increasing β for Libra+$ results in decreasing Job QoS Satisfaction, while Figure 3.5(b) shows that it results in increasing Cluster Profitability. This



(a) Job QoS Satisfaction



(b) Cluster Profitability

Figure 3.6: Impact of decreasing workload with dynamic pricing factor β.

demonstrates that the proposed pricing function is able to generate increasing profit even though a decreasing number of jobs is accepted. This is possible since jobs with sufficient budgets are executed at a higher cost (due to higher β) to compensate for accepting fewer jobs due to insufficient budgets. A point to note from Figure 3.5(a) is that if β is set too high, the Job QoS Satisfaction can be lower than other policies such as EDF-BF and FCFS-BF. An increasing β will also ultimately result in lower Cluster Profitability (eg. $\beta = 1.0$ in Figure 3.5(b)) since too many jobs are rejected by the high quoted prices.

These results demonstrate that the dynamic pricing factor β has a significant impact on both Job QoS Satisfaction and Cluster Profitability. A higher β lowers the level of sharing (a lower Job QoS Satisfaction), but increases the economic benefit of the cluster owner (a higher Cluster Profitability). However, β should not be set too high such that more jobs will be rejected and result in lower profit. Thus, the cluster owner can determine the level of sharing by adjusting the value of β. This demonstrates the flexibility of the pricing function in enabling the cluster owner to easily configure and determine the level of sharing based on his objective.

Figure 3.6 presents the impact of decreasing workload on the dynamic pricing factor β. Figure 3.6(b) shows that Libra+$ always return a higher Cluster Profitability than Libra, thus demonstrating the effectiveness of its pricing function in computing pricing based on demand (higher pricing for higher workload). This can also be verified by observing Libra+$ achieving higher Cluster Profitability when the workload is high (arrival delay factor <= 0.5) in Figure 3.6(b), though it manages lower Job QoS Satisfaction in Figure 3.6(a). In particular, a higher β is still able to generate higher Cluster Profitability with lower Job QoS Satisfaction.

A larger increase in Cluster Profitability is also obtained for higher β as the workload decreases. For example, the Cluster Profitability in Figure 3.6(b) increases by 25% for $\beta = 0.5$ (from 32% when the arrival delay factor is 0.25 to 57% when the arrival delay factor is 1.0). On the other hand, there is only an increase of 17% in Cluster Profitability for $\beta = 0.1$ (from 23% when the arrival delay factor is 0.25 to 40% when the arrival delay factor is 1.0). But again, if β is set too

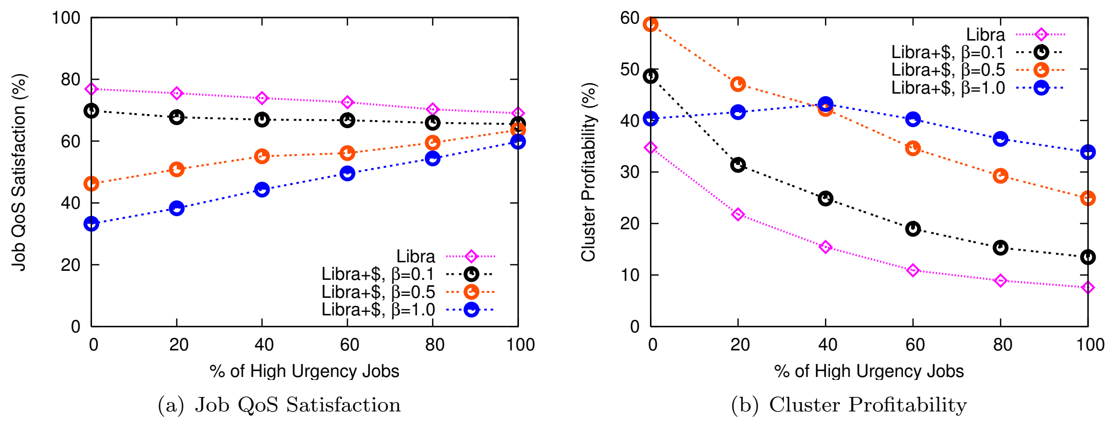


Figure 3.7: Impact of increasing number of high urgency jobs with dynamic pricing factor β.

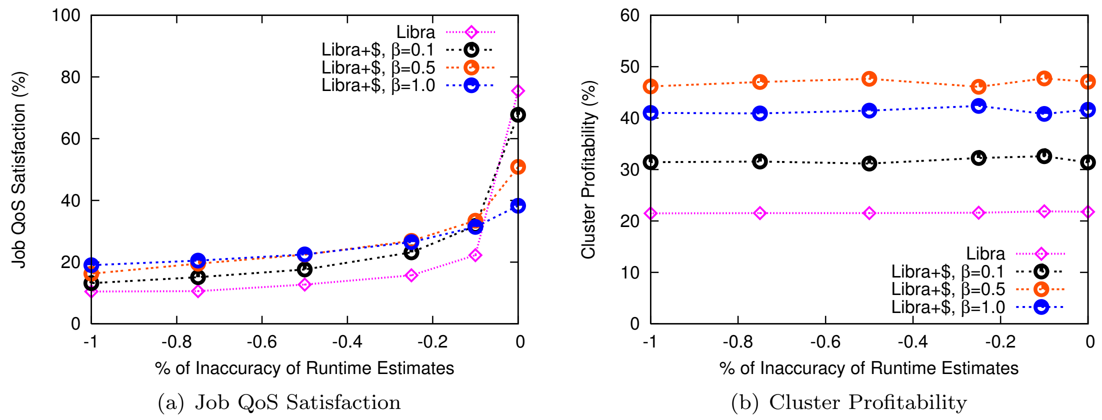


Figure 3.8: Impact of increasing runtime under-estimates with dynamic pricing factor β.

high (eg. $\beta = 1.0$ compared to $\beta = 0.5$), a smaller increase in Cluster Profitability is obtained instead (only 13% increase from 31% when the arrival delay factor is 0.25 to 44% when the arrival delay factor is 1.0).

However, Figure 3.7(b) shows that a high β (eg. $\beta = 1.0$) is able to generate higher Cluster Profitability when there are more high urgency jobs, but it can result in lower Cluster Profitability when there are less high urgency jobs. In addition, as seen in Figure 3.7(a), a high β also has higher Job QoS Satisfaction that gradually increases to almost equivalent to that of low β. This reinforces the need for a cluster owner to be aware of how the configuration of the dynamic pricing factor can easily influence the level of sharing and profit earned for his cluster.

3.6.3 Tolerance Against Runtime Under-estimates

Figure 3.8(a) shows that when there is no runtime under-estimates (0% of inaccuracy of runtime estimates), a higher dynamic pricing factor β for Libra+$ results in much lower Job QoS Satisfactions. But, with increasing under-estimates (reflected as decreasing negative % of inaccuracy of runtime estimates in Figure 3.8(a)), a higher β results in higher Job QoS satisfaction, whereas

Libra has the lowest Job QoS satisfaction. This demonstrates that a higher β provides a higher degree of tolerance against runtime under-estimates since fewer jobs are accepted and thus the possibility of delays occurring on later jobs is lower.

Figure 3.8(b) shows that increasing runtime under-estimates results in the lowest Cluster Profitability for Libra as fewer jobs are accepted due to delays caused by earlier jobs. However, a higher β for Libra+\$ can still maintain higher Cluster Profitability with increasing runtime under-estimates. This reiterates the effectiveness of the proposed pricing function in improving the economic benefit of the cluster owner, even in the case of runtime under-estimates.

3.7 Summary

We have demonstrated the importance of an effective pricing mechanism for achieving utility-based resource management and allocation in clusters, especially when demand exceeds supply of cluster resources. We show that our enhanced pricing function meets the four essential requirements for pricing of resources. In particular, our pricing function provides flexibility for the cluster owner to easily configure the pricing of his cluster resources to modify the level of sharing. Our pricing function also adapts to the changing supply and demand of resources by computing higher pricing when cluster workload increases. This serves as an effective means for admission control to prevent the cluster from overloading and tolerate against job runtime under-estimates. Finally, the pricing function generates a higher economic benefit for the cluster owner.

Chapter 4

Managing Penalties to Enhance Utility for Service Level Agreements

SLAs can be used to differentiate the importance or priority of jobs since they define service conditions that the cluster RMS agrees to provide for each different job. A SLA acts as a contract between a user and the cluster whereby the user is entitled to compensation whenever the cluster RMS fails to deliver the required service. This chapter outlines a simple SLA supporting four QoS parameters and presents a proportional processor share allocation technique called LibraSLA that takes into account any penalty incurred by accepting new jobs into the cluster based on their respective SLAs.

4.1 Related Work

Existing cluster RMSs such as Condor [115], LoadLeveler [70], Load Sharing Facility (LSF) [91], Portable Batch System (PBS) [10], and Sun Grid Engine (SGE) [108] still adopt system-centric approaches that optimize overall cluster performance. Cluster performance is often aimed at maximizing processor throughput and utilization for the cluster, and minimizing average waiting and response time for the jobs. They are thus not suitable for utility-based cluster computing since they do not differentiate and thus neglect varying levels of utility or value that different cluster users have for each job request. Maui [110] is an advanced cluster scheduler that is designed to be highly configurable and extensible. It can be extended to build customized user-level schedulers that incorporate fine-grained policies and examine numerous resource allocation parameters such as QoS and advanced reservation. Currently, no market-based approaches have been designed for Maui to improve utility for either the cluster or users.

Numerous market-based approaches [126] have been proposed for resource management in parallel and distributed computing. REXEC [43] is a remote execution environment for a cluster of workstations that adopts market-based resource allocation. It assigns resources proportionally to jobs based on their users' bid (valuation) for each job. Tycoon [76] also adopts the same bid-based proportional share technique as REXEC, but extends it with continuous bids for allocating resources in a Grid of distributed clusters. In contrast, our LibraSLA prioritizes each job based on its SLA parameters that address two additional perspectives: (i) deadline when a job has to be finished and (ii) penalty rate to compensate the user if the deadline is not met. In addition, we aim to improve the aggregate utility of the cluster through the consideration of penalties defined for respective SLA of different jobs.

Cluster-On-Demand [63] adopts distributed market-based task services to create a service market where penalties are incurred if jobs finish later than their required runtimes. It demonstrates the importance of balancing the reward against the risk of accepting and executing jobs, especially in the case of unbounded penalty. It also uses a discount rate based on present value to reduce future gains of a job in order to differentiate between delays in job execution. Similarly, our LibraSLA also considers penalties incurred on already accepted jobs by accepting a new job. But in our case, a job is penalized after the lapse of its deadline, instead of immediately after its runtime. In addition, we also determine which job has higher return so that the job with the highest return is assigned additional resources if available to realize its utility faster. LibraSLA also investigates resource allocation at a more fine-grained level as compared to Cluster-On-Demand. LibraSLA determines acceptance at the node level depending on available nodes within the cluster to execute a job. On the other hand, Cluster-On-Demand decides at the cluster level whether to accept a job into the cluster.

QoPS [64] is a QoS based scheduling technique for parallel jobs. It uses an admission control to guarantee the deadline of every accepted job by accepting a new job only if its deadline can be guaranteed without violating the deadlines of already accepted jobs. QoPS uses a slack factor for each job to represent the maximum delay that can be accommodated after its deadline. This allows earlier jobs with slack to be delayed if necessary so that future more urgent jobs can be accepted. On the other hand, our service model defines two types of deadlines: (i) hard deadline where the job has to be finished before the deadline and (ii) soft deadline where the job can finish anytime after the deadline. Instead of a slack factor, LibraSLA incorporates a SLA parameter called penalty rate to denote the user's flexibility with delays for soft deadlines through compensation. For the same job, a higher penalty rate means less flexibility than a lower penalty rate. Thus, LibraSLA attempts to minimize penalty to improve the cluster's aggregate utility. Another difference is that QoPS employs a kill-and-restart mechanism where a running job can be terminated to allow another job to be started so that a different schedule enables a new job to be accepted, while LibraSLA uses proportional share to vary the amount of resources for each job depending on their QoS needs.

A recent work [94] focuses on how service level objectives can be affected by penalty clauses defined in a SLA. It first outlines possible types of violations for a SLA, before identifying possible penalty clauses that can be included in a SLA to address these violations. Possible penalty clauses include decrease in the agreed payment for the service, additional compensation to the consumer, reduction in the future usage of the provider's service, and decrease in the reputation of the provider. LibraSLA combines both decrease in the agreed payment for the service and additional compensation to the consumer since the user pays less than its specified budget due to the penalty imposed after the lapse of deadline and is compensated once the penalty exceeds the budget. Another work [123] proposes specifying terms in a SLA as analytical functions instead of fixed values or range of values to increase the expressiveness of a SLA. Hence, there is a need to investigate the impact of more complex analytical functions as compared to the simple linear penalty function currently considered by LibraSLA.

Libra [102] is an earlier work done that successfully demonstrates that a market-based cluster scheduler is able to deliver more utility to users based on their QoS needs compared to traditional system-centric scheduling policies. It adopts the commodity market model by computing the price that users have to pay for their jobs be completed according to their QoS constraints. An enhanced pricing function [128] that is flexible, fair, dynamic and adaptive has also been proposed to improve the pricing scheme of the cluster so that the quoted price varies according to the workload of the cluster and prevents the cluster from overloading. LibraSLA incorporates a penalty function (through the penalty rate parameter in the SLA) where the utility or value of the user will decrease over time after the deadline of the job has lapsed. Libra assumes that all jobs have hard deadlines and guarantees that accepted jobs will be finished within their hard deadlines. In contrast, LibraSLA allows jobs with soft deadlines to be delayed and compensated to accommodate jobs with hard deadlines. Finally, Libra only accepts jobs based on the workload of the cluster, whereas LibraSLA also examines the return of accepting each new job with respect to the current aggregate utility and workload of the cluster.

4.2 SLA for Utility-based Cluster Computing

In utility-based cluster computing, clusters provide computing services to users who perceive varying utility or value for completion of jobs. Clusters need to have knowledge of the types of service demanded by different users for each job in order to prioritize jobs according to user's needs.

Clusters should thus support SLA that provides a means for users and the cluster to agree upon the service quality to be offered. In other words, SLA acts as a contract that outlines obligations that both users and the cluster have to enforce and fulfill. For example, users have to pay for the service provided, while the cluster needs to be penalized to compensate users for failing to offer the required service. This also means that users can negotiate with the cluster about the service quality to be provided before accepting the SLA.

Each job i submitted into the cluster has a specified SLA consisting of four QoS parameters:

1. deadline type $deadline\_type_i$: A deadline can be *hard* or *soft*. A hard deadline denotes that

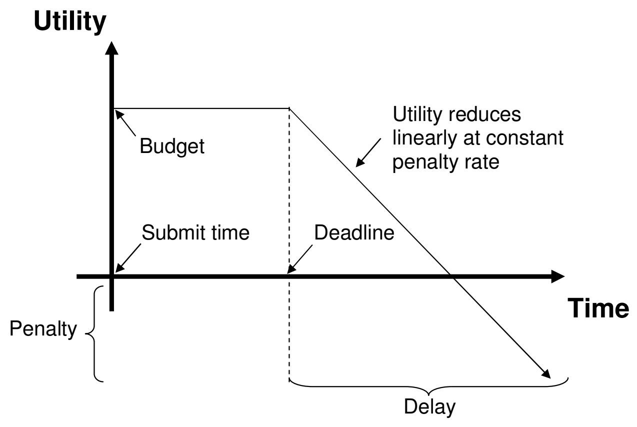


Figure 4.1: Impact of penalty function on utility.

the user wants the job to be finished before the deadline, whereas a soft deadline means that the user can accommodate delay. However, for soft deadline, the delay can be unbounded depending on the penalty rate of the job. Therefore, the user can provide an appropriate penalty rate value to possibly limit the maximium delay.

2. deadline D_i: The time period in which the job needs to be finished.

3. budget B_i: The maximum amount of currency that the user is willing to pay for the job to be completed.

4. penalty rate PR_i: The penalty rate penalizes the cluster to compensate the user for failure to meet the deadline of the job. It also reflects the user's flexibility with delayed deadline of the job. A higher penalty rate limits the delay to be shorter than that of a lower penalty rate.

Only these four QoS attributes are defined in our simple SLA since they are sufficient for our aim of studying how penalties can be managed in SLA-based resource allocations. We assume that the specified SLA accompanying each job submission has already been established through prior negotiations. Hence, we do not examine other research issues of SLA, such as the parties involved in establishing the SLA, the different phases of the SLA, and the specification of the SLA.

The key aspect of our SLA is that it incorporates a *penalty* function (through the penalty rate QoS parameter). This is realistic as users need to have assurance that their required services will be maintained by the cluster. The penalty function not only penalizes the cluster for its service failure, but compensates the users for tolerating the service failure. For simplicity, we model the penalty function as linear, as in other previous works [45][63]. Our penalty function reduces the budget of the job over time after the lapse of its deadline, rather than after its runtime in [63]. Figure 4.1 shows the impact of the penalty function on utility.

For each job i, the cluster achieves a utility U_i depending on its penalty rate PR_i and delay

DY_i:

$$U_i = B_i - (DY_i * PR_i) \tag{4.1}$$

Assuming that job i has a specified deadline D_i which starts from the time TSU_i when job i is submitted into the cluster, job i will have a delay DY_i if it takes longer to complete than its deadline D_i:

$$DY_i = max(0, (TF_i - TSU_i) - D_i) \tag{4.2}$$

where TF_i is the time when job i is completed. This means that job i has no delay (ie. $DY_i = 0$) if it completes before the deadline, with the cluster achieving the full budget B_i as utility U_i. If there is a delay (ie. $DY_i > 0$), U_i drops linearly till it becomes negative and transform into a penalty (ie. $U_i < 0$) after it exceeds B_i.

Since penalties are unbounded depending on the penalty rate and delay of each job, the cluster needs to be careful about accepting new jobs into the cluster. Failure to deliver the required service due to accepting too many jobs may result in heavily penalized jobs that can dramatically erode previously achieved utility. Therefore, we develop a SLA-based proportional share technique called LibraSLA (described in Section 4.3) that considers the risk of penalties to improve aggregate utility for the cluster.

In addition to the SLA requirements, LibraSLA considers the following job details:

1. Runtime Estimate RE_i: The runtime estimate is the estimated time period needed to complete job i on a compute node provided that it is allocated the node's full proportion of processing power. Thus, the runtime estimate varies on nodes of different hardware and software architecture and does not include any waiting time and communication latency. The runtime estimate can also be expressed in terms of the job length in million instructions (MI).

2. number of processors $PROC_i$: The number of processors requested by job i. A sequential job will need only a single processor (ie. $PROC_i = 1$), while a parallel job will request multiple processors (ie. $PROC_i > 1$).

4.3 LibraSLA: SLA-based Proportional Share with Utility Consideration

We consider utility-based resource management and allocation in a cluster with the following assumptions:

- Users express utility as the budget or amount of real money (as in the human world) they are willing to pay for the service. Real money is a well-defined currency [103] that will promote resource owner and user participation in distributed system environments. A user's budget is limited by the amount of currency that he has which may be distributed and administered through monetary authorities such as GridBank [20]. Since our focus is on resource allocation (which job to accept and which nodes to execute the accepted job), we do not venture further into other market concepts such as user bidding strategies and auction pricing mechanisms.

- Users only gain utility and thus pay for the service (based on QoS parameters in SLA) upon the completion of their jobs. For simplicity, the user pays the full budget for a job to the cluster if its deadline is fulfilled. But, if a job is delayed, the cluster will achieve a reduced utility or incur a penalty depending on length of the delay.
- The estimated runtime of each job provided during job submission is accurate. Estimated runtimes can be predicted in advance based on means such as past execution history.
- The deadline provided by a user for the job must be more than its runtime; otherwise it is not accepted into the cluster.
- The SLA of a job does not change after job acceptance. This means that the QoS parameters specified by the user during job submission do not change after the job is accepted.
- Users can only submit jobs into the cluster through the cluster RMS. This means that the cluster RMS has full knowledge of allocated workload currently in execution and remaining available resources on each compute node.
- The compute nodes can be homogeneous (have the same hardware architectures) or heterogeneous (have different hardware architectures). In the case of heterogeneous compute nodes, the estimated runtime needs to be converted to its equivalent on the allocated compute node.
- The operating system at each compute node uses time-shared scheduling support where multiple processor time partitions can be assigned to different jobs.

Bid-based proportional share [43][76] allocates proportions of a resource such as processor time to users based on their bids (budget QoS parameter in SLA) for each job. In other words, the resource share assigned to a job is proportional to the user's bid value in comparison to other users' bids. However, this approach does not take into consideration the characteristics of the job and its other essential SLA properties such as deadline and penalty rate.

To address this for SLA support, LibraSLA adopts the proportional share approach in Libra [102] that allocates processor time share $share_{ij}$ to job i on node j based on its current expected remaining runtime estimate $remainRE_{ij}$ and current remaining deadline QoS $remainD_i$:

$$share_{ij} = \frac{remainRE_{ij}}{remainD_i} \tag{4.3}$$

Assuming that a new submitted job i has a runtime estimate RE_i and deadline QoS D_i, job i will then have a current expected remaining time period of $remainRE_{ij}$ to execute depending on the amount of execution time already completed for job i on node j and a current remaining time period of $remainD_i$ before the deadline of job i expires depending on the amount of lapsed time since job i is submitted. At the time of job submission, $remainRE_{ij} = RE_i$ and $remainD_i = D_i$. $share_{ij}$ also denotes the minimum share that is required by job i in order to enforce its deadline D_i. Proportional share based on deadline not only allows more jobs to be accepted (since the allocated processor time shares are spread across the deadlines), but also ensures that their deadlines are met.

Therefore, the total processor time share $total\_share_j$ required to meet all deadlines of n_j jobs allocated to node j is:

$$total\_share_j = \sum_{i=1}^{n_j} share_{ij} \tag{4.4}$$

Delays occur when $total\_share_j$ exceeds the maximum processor time that node j can offer.

4.3.1 Computing Return for Jobs and Nodes

In order to improve aggregate utility for the cluster, LibraSLA needs to consider the utility of each job to determine which job has a higher return. The return $return_{ij}$ of a job i allocated to run on node j is computed as:

$$return_{ij} = (U_{ij}/RE_i)/D_i \tag{4.5}$$

Recall that it is possible for a job i allocated to node j to have negative utility (ie. $U_{ij} < 0$), also known as a penalty as defined in (4.1). Thus, in this case, job i will also have negative return (ie. $return_{ij} < 0$).

LibraSLA regards jobs that have shorter deadlines for an expected utility per unit of runtime estimate (U_{ij}/RE_i) to have a higher return. Jobs with shorter deadlines require a shorter commitment period as compared to those with longer deadlines. Thus, it increases the flexibility of accepting later arriving but possibly jobs with higher return as a full schedule of jobs with long deadlines may result in these future jobs being blocked by the admission control.

Jobs with shorter deadlines are also penalized more heavily than those with longer deadlines. This discourages accepting more jobs that can delay other accepted urgent jobs and jeopardize the cluster's aggregate utility.

LibraSLA can thus compute the return $return_j$ of node j to determine whether node j is improving the aggregate utility of the cluster or not:

$$return_j = \sum_{i=1}^{n_j} return_{ij} \tag{4.6}$$

$return_j$ also provides an indication of whether node j is overloaded with too many jobs and failing to satisfy their SLA. $return_j$ will be lower when the workload on node j is higher since insufficient resources will result in jobs being delayed and thus having lower utility.

4.3.2 Admission Control and Resource Allocation

Since the SLA of each job incorporates a penalty function (as described in Section 4.2), LibraSLA implements an admission control to ensure that more utility is achieved, instead of less utility due to accepting too many jobs and failing to meet their deadlines. The admission control determines whether a new job *new* should be accepted into the cluster depending on:

- $PROC_{new}$: A new job is not accepted if there are not enough available processors to run it. This often happens to parallel jobs as they require more processors.
- $deadline\_type_{new}$: If the new job requires hard deadline and there are no nodes that can fulfill its deadline, then it is not accepted.
- $return_j$: This denotes whether each node j will increase or decrease the aggregate utility if it is allocated this new job. Therefore, a new job can be accepted into the cluster depending on the return of each individual node.

Figure 4.2: Pseudocode for admission control and resource allocation of LibraSLA.

```
 1 for $j \leftarrow 0$ to $m-1$ do
 2     add job $new$ temporarily into $ListJobs_j$ ;
 3     $new\_return_j \leftarrow$ compute_new_return ($ListJobs_j$);
 4     remove job $new$ from $ListJobs_j$ ;
 5     if $new\_return_j \geq return_j$ then
 6         if $deadline\_type_{new}$ is $SOFT$ then
 7             place node $j$ in $ListHigherReturnNodes_{new}$ ;
 8         else if $deadline\_type_{new}$ is $HARD$ and $DY_{new} \leq 0$ then
 9             place node $j$ in $ListHigherReturnNodes_{new}$ ;
10         end
11     end
12 end
13 if $ListHigherReturnNodes_{new}\_size \geq PROC_{new}$ then
14     sort $ListHigherReturnNodes_{new}$ by $new\_return_j$ in descending order;
15     for $j \leftarrow 0$ to $PROC_{new}-1$ do
16         allocate job $i$ to node $j$ of $ListHigherReturnNodes_{new}$ ;
17     end
18 else
19     reject job $new$;
20 end
```

Figure 4.2 shows how LibraSLA decides whether to accept a new job based on nodes with the highest return. Assuming that the cluster has m nodes, LibraSLA first determines the return of each node (using compute_new_return function in Figure 4.3) for accepting the new job *new* (line 2–4). A node is suitable if it has higher return after accepting the new job and can satisfy its hard deadline if required (line 5–11). The new job is then accepted if there are enough suitable nodes as requested (line 13) and allocated to the node with the highest return (line 14–17).

Figure 4.3 computes the new return of a node for accepting a new job. It first determines total processor time share required to fulfill the deadlines of its allocated jobs plus the new job (line 5). It also identifies the job with the highest return based on the budget (line 6–9). If there is any remaining unallocated processor time, the job with the highest return is given this

Figure 4.3: Pseudocode for compute_new_return function.

```
1  $new\_return_j \leftarrow 0$;
2  $total\_share_j \leftarrow 0$;
3  set first job in $ListJobs_j$ to be job with the highest return;
4  for $i \leftarrow 0$ to $ListJobs_j\_size - 1$ do
5      $total\_share_j \leftarrow total\_share_j + share_{ij}$;
6      $return_{ij} \leftarrow (B_i/RE_i)/D_i$;
7      if job $i$ has higher return then
8          set job $i$ to be job with the highest return;
9      end
10 end
11 if $total\_share_j \geq$ maximum processor time of node $j$ then
12     increase share of job with the highest return by remaining unallocated processor time;
13     for $i \leftarrow 0$ to $ListJobs_j\_size - 1$ do
14         $new\_return_j \leftarrow new\_return_j + (B_i/RE_i)/D_i$;
15     end
16 else
17     for $i \leftarrow 0$ to $ListJobs_j\_size - 1$ do
18         if job $i$ is job with the highest return or job $i$ has HARD deadline then
19             $new\_return_j \leftarrow new\_return_j + (B_i/RE_i)/D_i$;
20         else
21             decrease share of job $i$ by shortfall proportion of processor time;
22             compute $DY_i$;
23             compute $U_{ij}$;
24             $new\_return_j \leftarrow new\_return_j + (U_{ij}/RE_i)/D_i$;
25         end
26     end
27 end
28 return $new\_return_j$;
```

additional remaining share to realize its utility faster (line 11–12). In this case, the return of the node is computed with the utility of each job same as its budget (line 13–15). If there is insufficient processor time, only the job with the highest return and jobs with hard deadlines are not delayed (line 18–19), while the other jobs with soft deadlines shares the shortfall processor time proportionally (line 21). The return of these delayed jobs is then computed accordingly (line 22–24).

4.4 Performance Evaluation

This section explains the evaluation methodology and scenarios for performance evaluation.

4.4.1 Evaluation Methodology

We apply the same evaluation methodology in Section 3.5.1. But, the experiments use a subset of the last 1000 jobs in the SDSC SP2 trace (April 1998 to April 2002) version 2.2 from Feitelson's Parallel Workload Archive [2]. This 1000 job subset which represents about 2.5 months of the original trace has an average inter arrival time of 2276 seconds (37.93 minutes) and average runtime of 10610 seconds (2.94 hours), and requires an average of 18 processors. The IBM SP2 located at San Diego Supercomputer Center (SDSC) has 128 compute nodes, each with a SPEC rating of 168.

Table 4.1: Default simulation settings for evaluating LibraSLA.

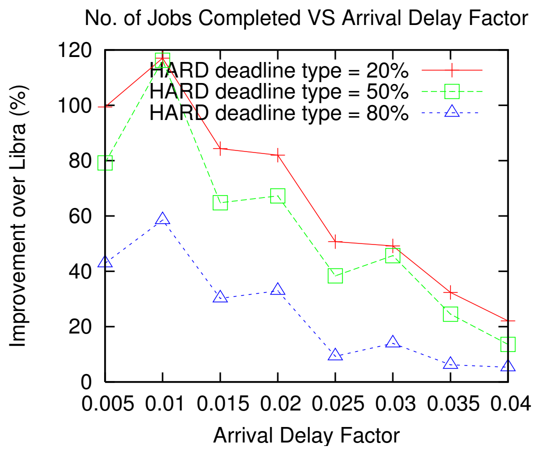
Parameter	**Default Value**
% of high urgency jobs	20.0
% of low urgency jobs	80.0
Deadline high:low ratio	7.0
Deadline low mean	3.5
Budget high:low ratio	7.0
Budget low mean	4.0
Penalty rate high:low ratio	4.0
Penalty rate low mean	4.0

We model two more SLA parameters deadline type and penalty rate, in addition to the deadline and budget parameters in Section 3.5.1 as follows. The penalty rate PR_i of each job i is modeled with respect to the real runtime R_i taken from the trace, similar to the deadline D_i and budget B_i. 20% of the jobs belong to a *high urgency* job class with a hard deadline of low D_i/R_i value, budget of high $B_i/f(R_i)$ value and penalty rate of high $PR_i/g(R_i)$ value, where $f(R_i)$ and $g(R_i)$ are functions to represent the minimum budget and penalty rate that the user will quote with respect to the real runtime R_i of job i. 80% of the jobs belong to a *low urgency* job class with a soft deadline of high D_i/R_i value, budget of low $B_i/f(R_i)$ value and penalty rate of low $PR_i/g(R_i)$ value. The penalty rate high:low ratio is modeled similarly to the budget high:low ratio as explained in Section 3.5.1.

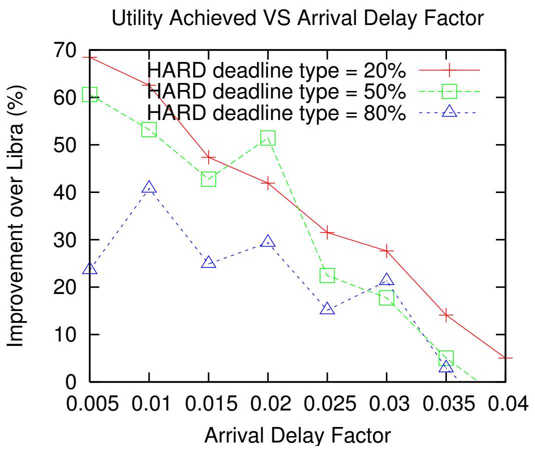
Table 4.1 lists the default settings for our simulations. We also use a mean factor to denote the mean value for the normal distribution of deadline, budget and penalty rate SLA parameters. A mean factor of 2 represents having mean value double than that of 1 (ie. higher).

4.4.2 Scenarios

Our performance evaluation examines the relative performance of LibraSLA with respect to Libra under varying cluster workload for the following SLA properties: (i) deadline type (Section 4.5.2), (ii) deadline (Section 4.5.3), (iii) budget (Section 4.5.4), and (iv) penalty rate (Section 4.5.5). Libra does not differentiate between hard and soft deadlines, thus accepting a new job only if there are sufficient nodes as requested to fulfill its deadline. Another key difference is that Libra selects nodes based on the best fit strategy. In other words, nodes that have the least available processor time after accepted the new job will be selected first. This ensures that nodes are saturated to their maximum so that more later arriving jobs may be accepted.



(a) No. of Jobs Completed



(b) Utility Achieved

Figure 4.4: Impact of deadline type on increasing workload.

4.5 Performance Results

In this section, we perform a detailed performance analysis of LibraSLA with respect to varying SLA properties: (i) deadline type, (ii) deadline, (iii) budget, and (iv) penalty rate.

4.5.1 Overview of Performance Results

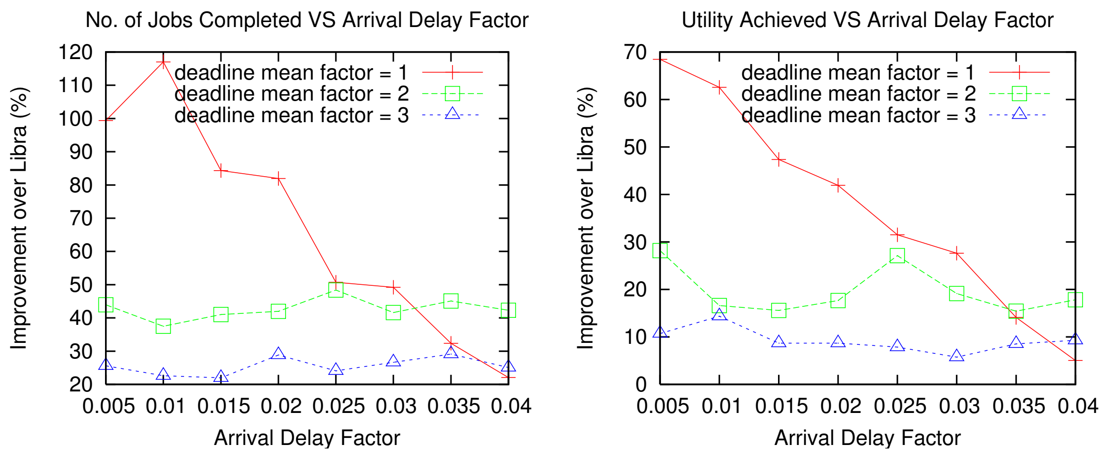
Generally, the improvement of LibraSLA over Libra decreases as the arrival delay factor increases. This is because Libra can also complete more jobs and achieve more utility when the workload is not heavy (higher arrival delay factor). Thus, we are able to demonstrate that LibraSLA is effective in differentiating jobs with higher utility in heavy workload situations.

4.5.2 Impact of Deadline Type

We vary the proportion of jobs belonging to the high urgency job class with hard deadlines at 20%, 50%, and 80% to examine the impact of deadline type on LibraSLA. Figure 4.4 shows that when there are more jobs with hard deadline (eg. 80%), the improvement over Libra is lower since there are less jobs with soft deadline to accommodate the required delays without risking the aggregate utility achieved. We can see that on average LibraSLA completes about 20% more jobs (Figure 4.4(a)) and achieves 10% more utility (Figure 4.4(b)) than Libra when there are 20% jobs with hard deadline as compared to 80%.

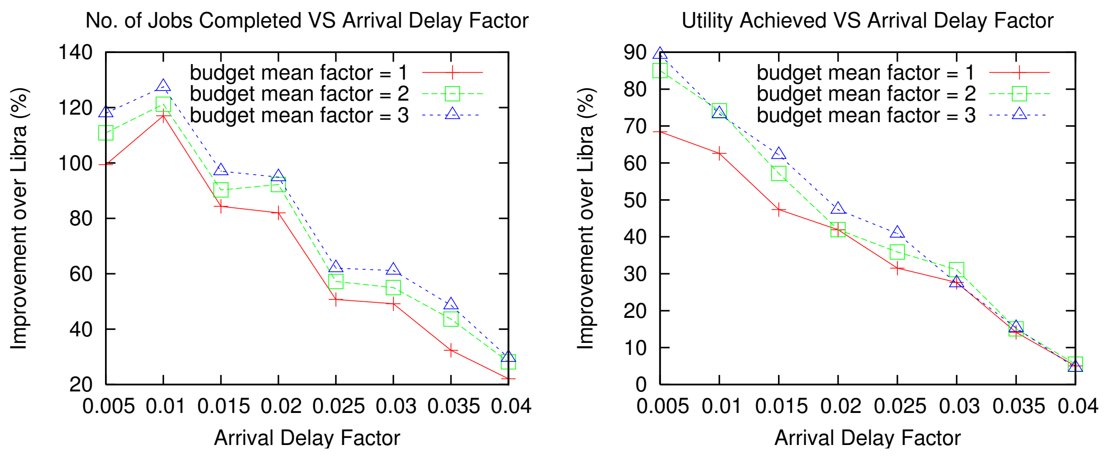
4.5.3 Impact of Deadline

We examine how LibraSLA performs for different deadlines using the deadline mean factor of 1, 2, and 3. Figure 4.5 shows that LibraSLA has a substantial large improvement over Libra for a deadline mean factor of 1 when the arrival delay factor is lower (0.005–0.025). This is because



(a) No. of Jobs Completed (b) Utility Achieved

Figure 4.5: Impact of deadline mean factor on increasing workload.



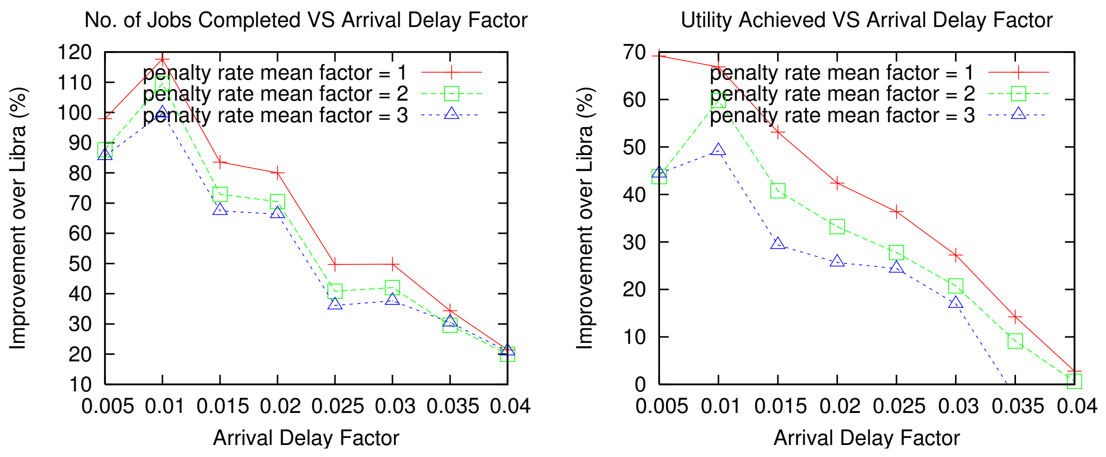
(a) No. of Jobs Completed (b) Utility Achieved

Figure 4.6: Impact of budget mean factor on increasing workload.

LibraSLA determines utility based on the deadline of the jobs, and thus can differentiate jobs with high utility and short deadline when the workload is high. However, with higher deadline mean factor of 2 and 3, the improvement over Libra is lower over increasing arrival delay factor since Libra is also able to complete more jobs with longer deadlines. We can also see that LibraSLA has a more constant improvement over Libra for sufficiently long deadlines (deadline mean factor of 2 and 3) as opposed to short deadlines (deadline mean factor of 1). This highlights that the deadline QoS has a strong impact on the performance of LibraSLA.

4.5.4 Impact of Budget

We investigate the performance of LibraSLA for different budgets with budget mean factor of 1, 2, and 3. Figure 4.6(a) shows that LibraSLA completes more jobs than Libra for higher budget mean factors. When jobs have higher budgets, LibraSLA can accommodate more soft deadline



(a) No. of Jobs Completed (b) Utility Achieved

Figure 4.7: Impact of penalty rate mean factor on increasing workload.

jobs with delays which in turn improves the aggregate utility (Figure 4.6(b)). However, the utility improvement also diminishes with increasing arrival delay factor (0.03–0.04) as Libra can also accept more jobs to increase utility.

4.5.5 Impact of Penalty Rate

We observe how LibraSLA performs with changing penalty rate mean factor of 1, 2, and 3. Figure 4.7 shows that LibraSLA has less improvement over Libra for increasing penalty rate mean factor. With higher penalty rate, LibraSLA is limited to accepting fewer jobs in order to preserve the aggregate utility. This explains the lower improvement in utility achieved since jobs with soft deadlines now has higher penalty rate and can potentially risk the aggregate utility.

4.6 Summary

This chapter has presented an approach to manage penalties incorporated in SLAs in order to enhance the utility of the cluster. We have also outlined a basic SLA with four QoS parameters: (i) deadline type, (ii) deadline, (iii) budget, and (iv) penalty rate, before describing a proportional share technique called LibraSLA that considers these QoS parameters. Simulation results demonstrate that LibraSLA performs better than Libra by accommodating more jobs through soft deadlines and minimizing penalties. This chapter has thus reinforced the need to employ and consider SLAs in cluster-level resource allocation in order to support utility-based cluster computing for service-oriented Grid computing.

Chapter 5

Managing Risk of Deadline Delay for Job Admission Control

Recent studies [102][63][64][92] have shown that implementing job admission control is essential to maintain the service performance of a cluster since a cluster has limited resources and cannot meet unlimited demand of resources from all users. A job admission control accepts a limited number of jobs so that the overall service performance of the cluster does not deteriorate. However, it is highly dependent on accurate runtime estimates of jobs in order to prioritize jobs effectively. Given that user runtime estimates may be rather inaccurate [84][113], this chapter examines how this inaccuracy can affect deadline constrained job admission control in clusters. This chapter also presents how an enhancement called LibraRiskD can manage the risk of inaccurate runtime estimates better by considering the risk of deadline delay.

5.1 Related Work

Existing cluster RMSs [115][70][91][10][108] neither consider SLAs for resource allocation nor implement job admission control, and thus are not able to enable service-oriented computing. Therefore, several works [102][63][64][92] have proposed implementing job admission control to support service-oriented computing.

In [63] and [92], their job admission controls aim to improve the overall utility or profit of service providers and do not consider the deadline QoS in their service requirements. Both Libra [102] and QoPS [64] use deadline constrained job admission control to fulfill the deadline QoS of jobs. Libra enforces hard deadlines of jobs, while QoPS allows soft deadlines by defining a slack factor for each job so that earlier jobs can be delayed up to the slack factor if necessary to accommodate later more urgent jobs. Instead, we focus on enforcing hard deadlines and thus enhance Libra to manage the risk of inaccurate runtime estimates.

Various works [72][73][63][92] have addressed some form of risk in computing jobs. In [63] and

[92], job admission control minimizes the risk of paying penalties to compensate users that will reduce the profit of service providers. Computation-at-Risk (CaR) [72][73] determines the risk of completing jobs later than expected based on either the makespan (response time) or the expansion factor (slowdown) of all jobs in the cluster. In contrast, our work examines the risk of inaccurate runtime estimates on job admission control to enforce the deadline QoS of jobs.

Some other works [84][100][113] have examined how inaccurate runtime estimates affect job scheduling in general, but not in deadline constrained job admission control for service-oriented computing.

5.2 LibraRiskD: Managing Risk for Job Admission Control

We consider deadline constrained job admission control in a cluster under the following scenario:

- The SLA requirements of a job do not change once the job has been accepted by the job admission control. We focus on one SLA requirement which is fulfilling the specified deadlines of jobs. Users specify hard deadlines so that jobs must be completed within deadlines to be useful.
- The cluster RMS supports non-preemptive job scheduling and provides the only single interface for users to submit jobs in the cluster. This means that it is aware of all workloads in the cluster.
- Each job i has the following characteristics:
  - Runtime Estimate RE_i: The runtime estimate is the estimated time period needed to complete job i on a compute node provided that it is allocated the node's full proportion of processing power. Thus, the runtime estimate varies on nodes of different hardware and software architecture and does not include any waiting time and communication latency. The runtime estimate can also be expressed in terms of the job length in million instructions (MI).
  - number of processors $PROC_i$: The number of processors requested by job i. A sequential job will need only a single processor (ie. $PROC_i = 1$), while a parallel job will request multiple processors (ie. $PROC_i > 1$).

In this section, we propose how the risk of deadline delay can be modeled and incorporated into the enhanced version of Libra called LibraRiskD to manage the risk of deadline delay.

5.2.1 Modeling Risk of Deadline Delay

Given that a job i has a specified deadline D_i which starts from the time TSU_i when job i is submitted into the cluster, job i incurs a delay DY_i if it takes longer to complete than its deadline

D_i:

$$DY_i = max(0, (TF_i - TSU_i) - D_i) \tag{5.1}$$

where TF_i is the time when job i is finished. Otherwise, job i has no delay (i.e. $DY_i = 0$s) and its deadline is fulfilled if it completes before the deadline.

The deadline constrained job admission control aims to maximize the number of jobs completed within their deadlines to enforce their SLAs, i.e. ideally, all accepted jobs are to be completed without any delay. Analogous to the CaR approach [72][73], we define a *deadline delay* metric DDY_i to model the impact of delay on the current remaining deadline $remainD_i$ of job i which is the remaining time period before the deadline of job i expires depending on the amount of lapsed time since job i is submitted:

$$DDY_i = \frac{DY_i + remainD_i}{remainD_i} \tag{5.2}$$

The minimum and best value of DDY_i is 1 when job i has zero delay. DDY_i has a higher impact value when DY_i is longer or $remainD_i$ is shorter. For example, job 1 with $DY_1 = 20$s and $remainD_1 = 5$s has $DDY_1 = 5$ which is higher, compared to job 2 with the same $DY_2 = 20$s and $remainD_2 = 10$s having $DDY_2 = 3$. This metric therefore discourages incurring longer job delays and violating deadlines of more urgent jobs. We can then compute the mean deadline delay μ_j of a node j that has n_j scheduled jobs:

$$\mu_j = \frac{\sum_{i=1}^{n_j} DDY_{ij}}{n_j} \tag{5.3}$$

Next, we determine the risk of deadline delay σ_j on node j by deriving its standard deviation:

$$\sigma_j = \sqrt{\frac{\sum_{i=1}^{n_j} (DDY_{ij})^2}{n_j} - (\mu_j)^2} \tag{5.4}$$

A high risk σ_j indicates a high uncertainty of jobs on node j not to experience deadline delays. Thus, not having any risk is the most ideal (i.e. $\sigma_j = 0$).

5.2.2 Managing Risk of Deadline Delay

LibraRiskD uses the same deadline-based proportional processor share technique as Libra [102] to determine the processor time allocation for each job on a node. Given that a job i still has a current expected remaining runtime estimate $remainRE_{ij}$ (initially its runtime estimate RE_i) on node j and has to be completed within a current remaining deadline $remainD_i$ (initially its deadline D_i), Libra computes the minimum processor time share $share_{ij}$ required as:

$$share_{ij} = \frac{remainRE_{ij}}{remainD_i} \tag{5.5}$$

Node j thus needs to have the total processor time share $total\_share_j$ to fulfill the deadlines of all its n_j jobs:

$$total\_share_j = \sum_{i=1}^{n_j} share_{ij} \tag{5.6}$$

Hence, in Libra, a new job is only accepted into node j if node j has at least $total\_share_j$ (including the new job) of processor time available. Otherwise, all the jobs on node j will encounter delays. If accepted, the new job starts execution immediately based on its allocated share $share_{ij}$.

Like Libra, the job admission control of LibraRiskD enforces the deadlines of previously accepted jobs on each node by rejecting a new job if there are not enough processors as required to execute it based on current workload. However, LibraRiskD applies two enhancements that differ from Libra. First, in Libra, a node j is suitable if it has at least $total\_share_j$ of processor time available, i.e. node j can meet the deadlines of all its jobs (including the new job). On the other hand, in LibraRiskD, a node j is suitable to accept a new job if its risk of deadline delay σ_j is zero. Second, Libra assigns the most suitable nodes to the new job based on the best fit strategy, i.e. nodes that have the least available processor time after accepting the new job will be selected first so that nodes are saturated to their maximum. In contrast, LibraRiskD only selects nodes that have zero risk of deadline delay.

Figure 5.1: Pseudocode for job admission control of LibraRiskD.

```
1  for $j \leftarrow 0$ to $m - 1$ do
2      add job $new$ temporarily into $ListJobs_j$ ;
3      for $i \leftarrow 0$ to $ListJobs_j\_size - 1$ do
4          determine $DY_i$;
5      end
6      compute $\sigma_j$;
7      remove job $new$ from $ListJobs_j$ ;
8      if $\sigma_j = 0$ then
9          add node $j$ into $ListZeroRiskNodes_{new}$ ;
10     end
11 end
12 if $ListZeroRiskNodes_{new}\_size \geq PROC_{new}$ then
13     for $j \leftarrow 0$ to $PROC_{new} - 1$ do
14         allocate job $new$ to node $j$ of $ListZeroRiskNodes_{new}$ ;
15     end
16 else
17     reject job $new$;
18 end
```

Figure 5.1 outlines how LibraRiskD works. Given that there are m compute nodes in the cluster, LibraRiskD first determines the delay that will be incurred on all jobs (including accepted jobs and the new job new) on each node j if job new is scheduled on it (line 2–5). The delay of a job on node j is determined based on the deadlines and runtime estimates of all the jobs on node j as explained in Section 5.2.1 (line 4). LibraRiskD then computes the risk of deadline delay σ_j for each node j (line 6). A node is suitable if it has zero risk after accepting job new (line 8–10). Job new is finally accepted and allocated to these suitable nodes if there is $PROC_{new}$ number of suitable nodes as required by job new; else it is rejected (line 12–18).

Table 5.1: Default simulation settings for evaluating LibraRiskD.

Parameter	Default Value
% of high urgency jobs	20.0
% of low urgency jobs	80.0
Deadline high:low ratio	4.0
Deadline low mean	4.0
Arrival delay factor	1.0
% of inaccuracy (accurate)	0.0
% of inaccuracy (actual)	100.0

5.3 Performance Evaluation

This section describes the evaluation methodology and scenarios for the experiments.

5.3.1 Evaluation Methodology

We use the same evaluation methodology in Section 3.5.1. But, the experiments use a subset of the last 3000 jobs in the SDSC SP2 trace (April 1998 to April 2002) version 2.2 from Feitelson's Parallel Workload Archive [2]. The SDSC SP2 trace is chosen as most other traces contain no information about actual runtime estimates of jobs. In addition, it has the highest resource utilization of 83.2% among other traces and thus ideally model the heavy workload scenario where having a job admission control can prevent overloading of the cluster. This 3000 job subset which represents about 2.5 months of the original trace has an average inter arrival time of 2131 seconds (35.52 minutes) and average runtime of 8880 seconds (2.47 hours), and requires an average of 17 processors. The IBM SP2 located at San Diego Supercomputer Center (SDSC) has 128 compute nodes, each with a SPEC rating of 168.

Table 5.1 lists the default settings for our simulations. The *inaccuracy* of runtime estimates is measured with respect to the actual runtime estimates of jobs obtained from the trace. An inaccuracy of 0% assumes runtime estimates are accurate and equal to the real runtime of the jobs. On the other hand, an inaccuracy of 100% means that runtime estimates are the actual runtime estimates as reflected in the trace which may be accurate (equal to the real runtime) or inaccurate (not equal to the real runtime).

5.3.2 Scenarios

First, we assess job admission control based on three different scenarios: (i) varying workload (Section 5.4.2), (ii) varying deadline high:low ratio (Section 5.4.3), and (iii) varying high urgency jobs (Section 5.4.4). For each scenario, we compare how various job admission controls perform for both accurate runtime estimates and actual runtime estimates from the trace. The accurate runtime estimates reflect the ideal performance of each control and show how they differ with actual runtime estimates from the trace. Then, we evaluate based on varying inaccurate runtime estimates for both 20% and 80% of high urgency jobs (Section 5.4.5). This will demonstrate whether LibraRiskD can manage the risk of inaccurate runtime estimates more effectively.

We examine the performance of each scenario based on two metrics: (i) percentage of jobs with deadlines fulfilled, and (ii) average slowdown. The percentage of jobs with deadlines fulfilled is the number of jobs that are completed within their specified deadlines, out of the total number of jobs submitted. The slowdown SD_i of a job i is the ratio of its response time and minimum runtime required, where the response time is the time taken for job i to be completed after it is submitted and includes waiting time. The average slowdown of n number of jobs is then computed as $(\sum_{i=1}^{n} SD_i)/n$. Since the emphasis of our work is to meet the deadlines specified for jobs, for all our experiments, the average slowdown is computed only for jobs with deadlines fulfilled.

We also implement a non-preemptive Earliest Deadline First backfilling (EDF-BF) strategy to facilitate comparison with Libra and LibraRiskD. Unlike Libra and LibraRiskD, EDF-BF executes only a single job on a processor at any time (i.e. space-shared) and maintains a single queue to store incoming jobs. EDF-BF selects the job with the earliest deadline to execute first and uses EASY backfilling [79][84] to assign unused processors to the next waiting jobs in the queue provided that they do not delay the first job based on their runtime estimates.

We find that EDF-BF without job admission control performs much worse as compared to with job admission control, especially when deadlines of jobs are short. But, unlike Libra and LibraRiskD, we incorporate a more relaxed job admission control for EDF-BF where a job is not rejected immediately during job submission. Instead, EDF-BF only rejects a selected job prior to execution if its deadline has expired or its deadline cannot be met based on its runtime estimate. In other words, EDF-BF can reselect a new job with an earlier deadline that arrives later during the waiting phase to improve its selection choice. Therefore, EDF-BF has two advantages over Libra and LibraRiskD: more generous job admission control and better selection choice.

5.4 Performance Results

This section first explains the similar results that are observed for the following three scenarios: (i) varying workload, (ii) varying deadline high:low ratio, and (iii) varying high urgency jobs. It then differentiates the performance for each of these scenarios, and varying inaccurate runtime estimates.

5.4.1 Overview: Varying Workload, Deadline High:Low Ratio, and High Urgency Jobs

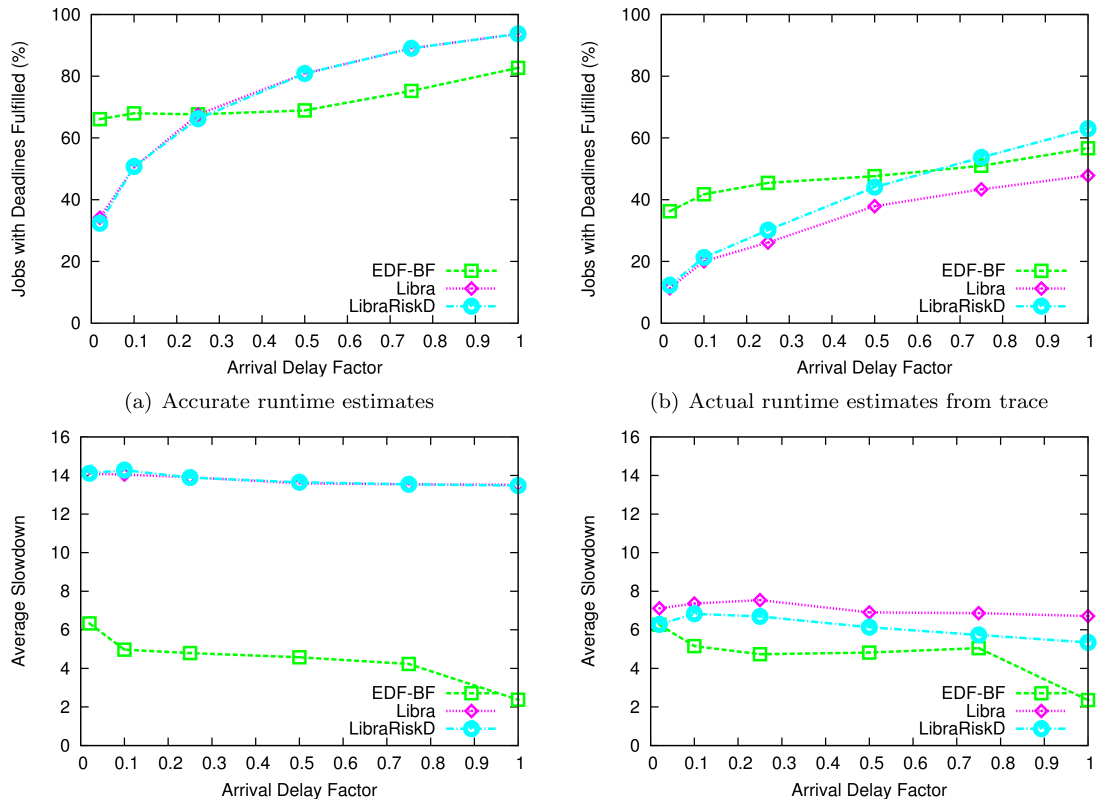
This overview explains similar results that are observed for all three scenarios (Figure 5.2, 5.3, and 5.4). Overall, more jobs are completed with their deadlines fulfilled for the ideal case of accurate runtime estimates than the case of using actual runtime estimates from the trace. This is because in practice, runtime estimates are often overestimated during job submission to reduce the possibility of jobs being terminated due to inadequate request of runtime. Thus, all three job admission controls will actually accept fewer jobs than expected, leading to fewer jobs with deadlines fulfilled.

With the assumption of accurate runtime estimates, Libra is able to fulfill more jobs within their deadlines than EDF-BF (Figure 5.2(a), 5.3(a), and 5.4(a)). One reason is that Libra ensures that the deadline of a job can be fulfilled based on its runtime estimate before accepting it. Another reason is that Libra adopts the best fit strategy so that nodes are saturated to their maximum to accommodate more jobs.

But, we can see that Libra performs worse than EDF-BF with the use of actual runtime estimates from the trace (Figure 5.2(b), 5.3(b), and 5.4(b)). This exposes the core weakness in Libra as it relies heavily on the idealistic assumption of accurate runtime estimates. In contrast, LibraRiskD can fulfill many more jobs than Libra for actual runtime estimates from the trace (Figure 5.2(b), 5.3(b), and 5.4(b)), while fulfilling as many jobs as Libra for accurate runtime estimates (Figure 5.2(a), 5.3(a), and 5.4(a)).

We now compare the results for average slowdown. Both Libra and LibraRiskD incur the same average slowdown for accurate runtime estimates (Figure 5.2(c), 5.3(c), and 5.4(c)). However, LibraRiskD achieves lower average slowdown than Libra for actual runtime estimates from the trace (Figure 5.2(d), 5.3(d), and 5.4(d)). EDF-BF has the lowest average slowdown that remains unchanged for both cases because in our evaluation methodology, the deadline D_i of a job i is always set as a factor higher than its real runtime R_i from the trace via D_i/R_i. Thus, EDF-BF is implicitly executing smaller jobs first based on the deadlines. This is also why the average slowdown of EDF-BF only marginally increases for increasing deadline high:low ratio (Figure 5.3(c) and 5.3(d)) and slightly drops for increasing number of high urgency jobs (Figure 5.4(c) and 5.4(d)).

Therefore, LibraRiskD is not only able to perform comparatively well as Libra based on accurate runtime estimates, but is also capable of fulfilling many more jobs, and achieve lower average slowdown than Libra based on actual runtime estimates from the trace. This highlights the importance of considering the risk of deadline delay and clearly illustrates the effectiveness of LibraRiskD in doing so.



(a) Accurate runtime estimates
(b) Actual runtime estimates from trace
(c) Accurate runtime estimates
(d) Actual runtime estimates from trace

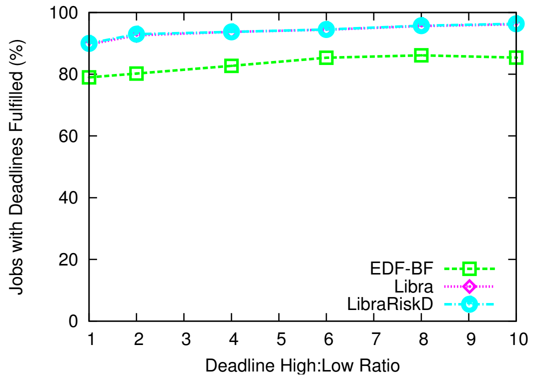
Figure 5.2: Impact of varying workload.

5.4.2 Varying Workload

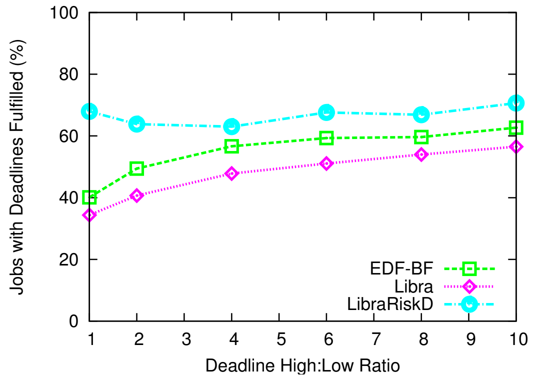
In Figure 5.2, we vary the arrival delay factor (from 0 to 1) to depict decreasing workload. As the workload decreases, increasing number of jobs are completed with deadlines fulfilled (Figure 5.2(a) and 5.2(b)), with decreasing (improving) average slowdown (Figure 5.2(c) and 5.2(d)). For accurate runtime estimates, when the workload is heavy (arrival delay factor < 0.3), EDF-BF fulfills more jobs within their deadlines than Libra and LibraRiskD (Figure 5.2(a)), but fulfills less jobs as the workload decreases (arrival delay factor > 0.3).

Unlike Libra and LibraRiskD that determine whether to accept or reject a job immediately during job submission, EDF-BF maintains a single queue to store incoming jobs and thus do not reject jobs instantaneously. As EDF-BF waits for the requested number of processors to be available for a currently selected job, it can reselect a new job with an earlier deadline that arrives during the waiting period. Thus, EDF-BF has a more favorable selection choice during heavy workload as more jobs arrive and improve the selection choice leading to EDF-BF fulfilling more jobs than Libra and LibraRiskD. However, this unfair advantage diminishes when the arrival delay factor is more than 0.3 as the selection choice decreases with increasing arrival delay factor.

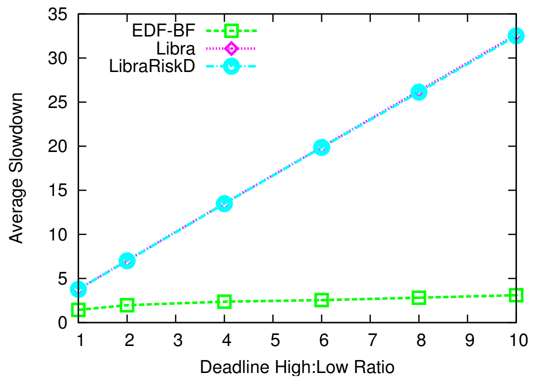
With actual runtime estimates from the trace, LibraRiskD progressively completes an increasing higher number of jobs with deadlines fulfilled (Figure 5.2(b)), with decreasing lower average



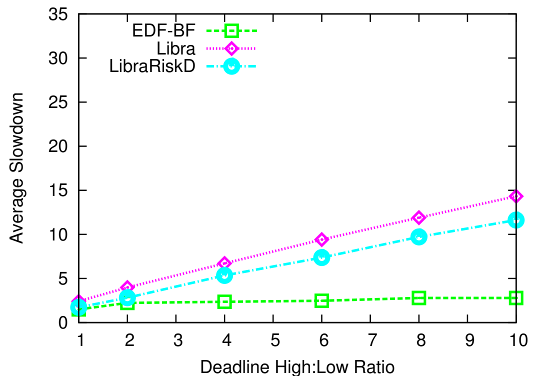
(a) Accurate runtime estimates



(b) Actual runtime estimates from trace



(c) Accurate runtime estimates



(d) Actual runtime estimates from trace

Figure 5.3: Impact of varying deadline high:low ratio

slowdown (Figure 5.2(d)) than Libra as the workload decreases (arrival delay factor > 0.5).

5.4.3 Varying Deadline High:Low Ratio

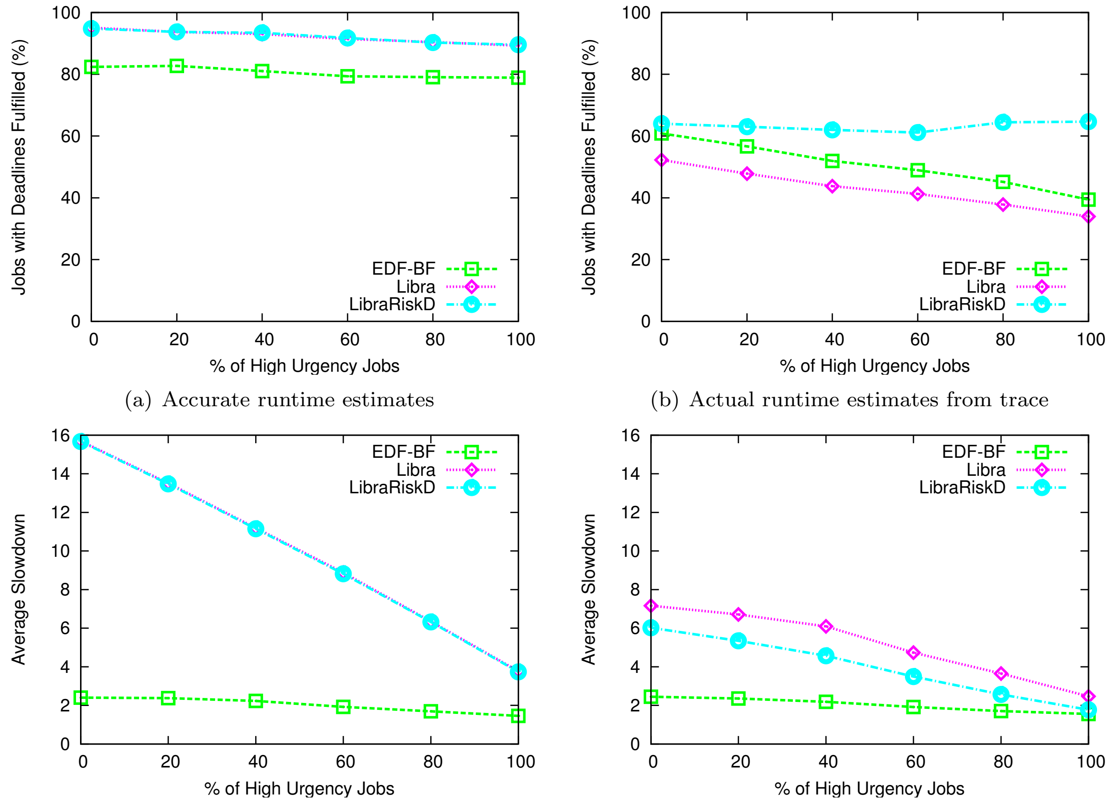
Figure 5.3 shows the impact of increasing deadlines for low urgency jobs (deadline high:low ratio from 1 to 10). With increasing deadlines, more jobs have their deadlines fulfilled (Figure 5.3(a) and 5.3(b)), but the average slowdown (Figure 5.3(c) and 5.3(d)) rises as longer jobs are accepted.

For the case of using actual runtime estimates from the trace, LibraRiskD finishes more jobs with deadlines fulfilled (Figure 5.3(b)) than Libra even though the improvement is higher when the deadline high:low ratio is low (< 4). LibraRiskD also attains a gradually improving lower average slowdown than Libra as deadlines increase (Figure 5.3(d)).

5.4.4 Varying High Urgency Jobs

Figure 5.4 shows the performance for various proportions of high urgency jobs (from 0% to 100%). As the number of high urgency jobs increases, fewer jobs are being accepted and completed, leading to decreasing number of jobs with deadlines fulfilled (Figure 5.4(a) and 5.4(b)) and decreasing average slowdown (Figure 5.4(c) and 5.4(d)).

With actual runtime estimates from the trace, LibraRiskD completes close to 30% more jobs



(a) Accurate runtime estimates

(b) Actual runtime estimates from trace

(c) Accurate runtime estimates

(d) Actual runtime estimates from trace

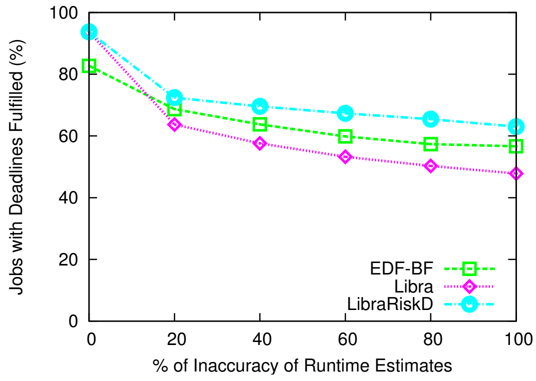
Figure 5.4: Impact of varying high urgency jobs.

than Libra where there is 100% high urgency jobs (Figure 5.4(b)), which is thrice better than the 10% improvement over Libra for 0% high urgency jobs. LibraRiskD also satisfies increasing number of jobs within their deadlines as the number of high urgency jobs increases, whereas both EDF-BF and Libra satisfy decreasing number of jobs. Hence, LibraRiskD can meet more jobs with shorter deadlines. It also maintains an improvement over Libra for average slowdown (Figure 5.4(d)).

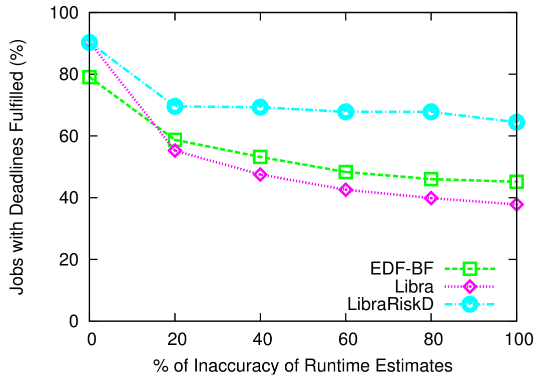
5.4.5 Varying Inaccurate Runtime Estimates

Figure 5.5 compares the difference between executing jobs with 20% and 80% of high urgency jobs in terms of varying inaccurate runtime estimates (from 0% to 100%). Generally, more jobs are completed with their deadlines fulfilled for 20% of high urgency jobs as compared to 80% of high urgency jobs since having more high urgency jobs results in fewer jobs being fulfilled. As such, the average slowdown is also lower (better) when there is 80% of high urgency jobs.

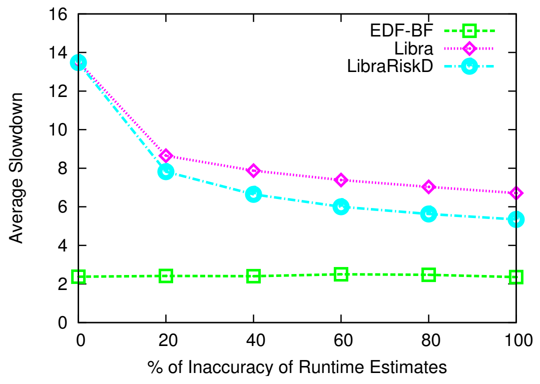
In Figure 5.5(a) and 5.5(b), decreasing number of jobs are completed with deadlines fulfilled as the inaccuracy of runtime estimates increases. But, LibraRiskD fulfills higher number of jobs than both EDF-BF and Libra. We can see that LibraRiskD completes much more (about twice as many) jobs when there is 80% of high urgency jobs as compared to 20% high urgency jobs. In fact, LibraRiskD still maintains similar number of jobs with deadlines fulfilled for the case of 80% high



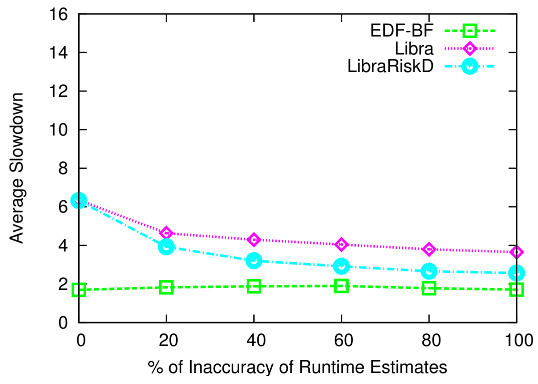
(a) 20% of high urgency jobs



(b) 80% of high urgency jobs



(c) 20% of high urgency jobs



(d) 80% of high urgency jobs

Figure 5.5: Impact of varying inaccurate runtime estimates.

urgency jobs (Figure 5.5(b)) as that of 20% high urgency jobs (Figure 5.5(a)), while both EDF-BF and Libra experience drops in numbers.

Figure 5.5(c) and 5.5(d) show that the average slowdown decreases for both Libra and LibraRiskD as the inaccuracy of runtime estimates increases. EDF-BF has the lowest average slowdown that remains the same for both cases as it also executes smaller jobs first based on their deadlines. Otherwise, as the inaccuracy of runtime estimates increases, LibraRiskD maintains similar improvement in average slowdown over Libra when there is 20% or 80% of high urgency jobs.

In short, LibraRiskD is able to fulfill more jobs for higher inaccuracies of runtime estimates as compared to Libra, especially when there are more high urgency jobs with shorter deadlines. This thus demonstrates its effectiveness in managing the risk of inaccurate runtime estimates.

5.5 Summary

This chapter reveals that deadline constrained job admission control performs worse than expected when using actual runtime estimates from traces of supercomputer centers because they rely on accurate runtime estimates, whereas actual runtime estimates are inaccurate and often over estimated. Therefore, we propose an enhanced deadline constrained job admission control called

LibraRiskD that can manage the risk of inaccurate runtime estimates more effectively via a deadline delay metric. Simulation results demonstrate that LibraRiskD completes the most number of jobs with deadline fulfilled as compared to EDF-BF and Libra in three cases when: (i) the cluster workload is lower, (ii) the deadlines of jobs are shorter (more urgent), or (iii) the runtime estimates are less accurate. LibraRiskD also achieves considerably lower average slowdown than Libra. This chapter has thus addressed the importance of managing the risk of inaccurate runtime estimates for deadline constrained job admission control in clusters.

Chapter 6

Risk Analysis of a Computing Service Provider

In utility computing, users only have to pay when they use the computing services. They do not have to invest on or maintain computing infrastructures themselves, and are not constrained to specific computing service providers. Thus, a computing service provider will face two new challenges: (i) what are the objectives or goals it needs to achieve in order to support the utility computing model, and (ii) how to evaluate whether these objectives are achieved or not. This chapter identifies four essential objectives that are required to support the utility computing model and describes two evaluation methods that are simple and intuitive to analyze the effectiveness of resource management policies in achieving the objectives.

6.1 Related Work

Numerous RMSs [115][70][91][10][108] are available to provide different policies to allocate jobs. However, new service parameters need to be considered and enforced for utility computing, such as the deadline to complete the job, the budget the user will pay for its completion, and the penalty for any deadline violation. Therefore, several new works [102][63][64][92][122][127][128] proposes policies using admission control to support quality-driven computing services by selectively accepting new jobs based on certain service parameters. Admission control helps to maintain the level of service when there is only a limited supply of computing resources to meet an unlimited demand of service requests. Hence, when the demand is higher than the supply of resources, a computing service needs to either reject new service requests to ensure previously accepted requests are not affected or compromise previously accepted service requests to accommodate new requests. But, there is no proposed work to evaluate the effectiveness of these service-oriented policies in the context of utility computing, in particular the ability to satisfy essential objectives of a computing service provider.

Table 6.1: Focus of four essential objectives.

Focus	**Objective**	**Abbreviation**
User-centric	manage wait time for SLA acceptance	*wait*
	meet SLA requests	*SLA*
	ensure reliability of accepted SLA	*reliability*
Provider-centric	attain profitability	*profitability*

Various other works [72][73][63][92] have addressed some form of risk in computing jobs. In [63] and [92], the risk of paying penalties to compensate users is minimized so as not to reduce the profit of service providers. Computation-at-Risk (CaR) [72][73] determines the risk of completing jobs later than expected based on either the makespan (response time) or the expansion factor (slowdown). GridIS [122] shows that a conservative provider earns much less profit due to accepting too few jobs to run, as compared to an aggressive provider who earn more profit even though more jobs result in deadline violations. However, none of these works consider and model the impact of policies on the achievement of objectives as risks.

Our work is inspired by service management [116] and risk management [47] in the field of economics which has been widely studied, adopted and proven. From the economics perspective, a computing service provider in utility computing is a business intended to sell services to consumers and generate profit from them. Comprehensive studies in service management [101] have shown that customer satisfaction is a crucial success factor to excel in the service industry. Customer satisfaction affects customer loyalty, which in turn may lead to referrals of new customers [116]. These achievements thus constitute the sustainability and improvement of revenue for a business. Therefore, we apply similar service quality factors for the proposed three user-centric objectives to ensure that customer satisfaction is achieved. In addition, economists have proposed enterprise risk management [83] to manage the risks of a business based on its targeted objectives. Hence, we adopt a similar approach to evaluate resource management policies of a computing service provider with respect to its objectives using separate and integrated risk analysis.

6.2 Objectives of a Computing Service Provider

This section explains why a computing service provider wants to achieve four essential objectives and how to measure these objectives. As listed in Table 6.1, the four objectives consists of three user-centric objectives: (i) manage wait time for SLA acceptance (*wait*), (ii) meet SLA requests (*SLA*), and (iii) ensure reliability of accepted SLA (*reliability*); and one provider-centric objective: (i) attain profitability (*profitability*).

A user-centric objective can influence service users, whereas a provider-centric objective can only affect computing service providers. However, in utility computing, a computing service provider also has to consider or even place greater emphasis on user-centric objectives. This is because a computing service provider has to be commercially viable and is thus heavily dependent on revenue generated by service users who pay and expect quality-driven and value-added services to be provided.

In addition, we believe that the utility computing model marks an important milestone for the creation of a free market economy to buy and sell computing resources based on actual usage. In this free market economy, we envision the availability of numerous computing service providers which have the required capability to process any specific job characteristics at any time. These computing services will thus actively compete with one another to increase their market share of service users so as to increase their revenue. This means that service users can switch to any computing service whenever they want. Therefore, ignoring user-centric objectives is likely to result in dwindling number of users, loss of reputation and revenue, and finally out-of-business for a computing service provider.

We consider all four objectives to be equally important and thus have the same priority as they address various operational aspects of a computing service provider, from accepting and fulfilling service requests ($wait$ and SLA objectives) to monitoring service levels and monetary yields (*reliability* and $profitability$ objectives). However, in the proposed integrated risk analysis (described in Section 6.3.2), we allow a computing service provider to prioritize objectives differently by adjusting the corresponding weight of each objective. Hence, this provides the flexibility for different computing service providers to control the objectives based on their specific interests.

For the measurement of the $wait$ objective (in Section 6.2.1), we decide to compute an average value for it. The average value provides the central tendency of wait times required for various jobs to be accepted. The minimum average value of the $wait$ objective is 0 time units.

But, for the measurement of SLA, *reliability*, and $profitability$ objectives (in Section 6.2.2, 6.2.3 and 6.2.4), we choose to compute a percentage value for each of them. Each percentage value provides a relative performance value with respect to the maximum upper bound value of a specific objective, which is more informative and meaningful, as compared to just having an absolute performance value. As an example, for the SLA objective, the total number of m jobs submitted to the computing service is the maximum number of jobs that can possibly have SLA fulfilled by the computing service. Measuring the percentage value of n_{SLA} jobs with SLA fulfilled out of the total number of m jobs submitted is thus more meaningful compared to just the value of n_{SLA}. The minimum and maximum percentage value of SLA, *reliability*, and $profitability$ objectives is 0% and 100% respectively.

6.2.1 Manage Wait Time for SLA Acceptance

Customers perceive the responsiveness of a business as a service quality factor because it reflects the willingness of the business to provide fast service and help customers quickly [101]. Moreover, time is a valuable and critical factor for an individual or organization to survive and excel in today's highly dynamic and competitive environment that demands continuous monitoring and quick response. In utility computing, a user submitting the service request for a job has to wait for the computing service provider to examine and accept the request before starting the actual processing of the job. Thus, we assume that every user accessing the computing service also requires timely processing of their requests in order to satisfy other personal or organizational commitments. In other words, a user who wastes greater time to secure a service request will be more disadvantaged as any delay will impact on the prompt completion of other commitments.

The process of managing the wait time for SLA acceptance can be accessed through the typical amount of time taken by the computing service provider to accept and execute jobs. Hence, the *wait* objective is measured as:

$$wait = \frac{\sum_{i=1}^{n_{SLA}} TST_i - TSU_i}{n_{SLA}} \tag{6.1}$$

where TST_i is the time when job i starts execution after being accepted, TSU_i is the time when job i is submitted to the computing service, and n_{SLA} is the number of jobs with SLA fulfilled. A lower value of $wait$ is better than a higher value.

6.2.2 Meet SLA Requests

Another service quality factor perceived by customers is the assurance of a business that it is adequately knowledgeable and sufficiently competent [101]. This inspires the trust and confidence of customers in the business. In utility computing, a computing service provider has to assure service users of its ability to satisfy service demand by meeting SLA requests.

An inability to meet service demand can be verified by a decrease in the number of requested SLAs that are fulfilled successfully. Therefore, the SLA objective is computed as:

$$SLA = \frac{n_{SLA}}{m} * 100 \tag{6.2}$$

where n_{SLA} is the number of jobs with SLA fulfilled and m is the number of jobs submitted to the computing service. A higher value of SLA is better than a lower value.

6.2.3 Ensure Reliability of Accepted SLA

The level of customer satisfaction for a business can be affected by the reliability of the business to deliver expected performance dependably and accurately [101]. In utility computing, since users specify the level of service they require through SLAs, a computing service provider wants to ensure that it is able to really deliver the agreed level of service by ensuring reliability of accepted SLAs. We assume that a computing service provider has monitoring mechanisms to check the progress of existing job executions and adjust resources accordingly to meet current and future service obligations.

A compromise in service quality can be ascertained by an increase in the number of accepted SLAs that are not fulfilled successfully. Thus, the *reliability* objective is calculated as:

$$reliability = \frac{n_{SLA}}{n} * 100 \tag{6.3}$$

where n_{SLA} is the number of jobs with SLA fulfilled and n is the number of jobs that are accepted by the computing service. A higher value of *reliability* is better than a lower value.

6.2.4 Attain Profitability

The most important objective for a computing service provider is to attain profitability as Return On Investment (ROI) for providing the service since commercial businesses are driven by monetary performance and thus need to track their monetary yields. We assume that a computing service provider has accounting and pricing mechanisms to record resource usage information and compute usage costs to charge service users accordingly.

The cost paid by the service users can also be viewed as the utility or ROI earned by the computing service provider. Hence, the *profitability* objective is determined as:

$$profitability = \frac{\sum_{i=1}^{n} U_i}{\sum_{i=1}^{m} B_i} * 100 \tag{6.4}$$

where $\sum_{i=1}^{n} U_i$ is the total utility earned from n jobs accepted by the computing service and $\sum_{i=1}^{m} B_i$ is the total budget of m jobs that are submitted to the computing service. A higher value of $profitability$ is better than a lower value. The $profitability$ objective is computed as a ratio of the total budget specified by the users for submitted jobs in order to reflect the potential of maximizing the ROI.

6.3 Risk Analysis

A computing service provider must now be able to assess whether its implemented resource management policy is able to achieve any or all of the objectives to support the utility computing model. This section proposes two evaluation methods that is derived from enterprise risk management [83]: (i) separate and (ii) integrated risk analysis. Both methods evaluate a policy using two indicators: (i) performance and (ii) volatility. Performance acts as the value measure of the policy, while volatility acts as the risk measure. Volatility is selected as the risk measure since it reflects how performance values fluctuate and thus the consistency of the policy in returning similar performance values. This section then describes how the level of associated risk can be easily visualized through risk analysis plots produced from these performance and volatility values.

6.3.1 Separate Risk Analysis

Separate risk analysis analyzes the performance and volatility involved in a single objective for a particular scenario. An example of a scenario is varying workload whereby only the workload changes while the rest of the experiment settings remains the same. Hence, measuring a specific objective for the varying workload scenario will return a total of r results with each result from a different workload.

However, the raw values measured for the four objectives do not constitute a consistent and correct outcome. As highlighted in Section 6.2, a lower value of the *wait* objective is better than a higher value, whereas a higher value of SLA, *reliability*, and $profitability$ objectives is better than a lower value. Therefore, we normalize these raw values accordingly to obtain normalized values that are standardized within the range 0 to 1, with the minimum value of 0 symbolizing the worst performance and the maximum value of 1 symbolizing the best performance respectively. It is important to note that a normalized value of 0.5 (as an example) for one metric does not necessarily correspond to a normalized value of 0.5 for another metric.

The performance μ_{sep} and volatility σ_{sep} for the separate risk analysis of an objective in a particular scenario can be computed as the mean of all r normalized results obtained in the scenario and standard deviation of these r normalized results respectively:

$$performance, \mu_{sep} = \frac{\sum_{i=1}^{r} normalized\_result_i}{r} \tag{6.5}$$

$$volatility, \sigma_{sep} = \sqrt{\frac{\sum_{i=1}^{r} (normalized\_result_i)^2}{r} - (\mu_{sep})^2} \tag{6.6}$$

where $0 \leq normalized\_result_i \leq 1$.

6.3.2 Integrated Risk Analysis

Since separate risk analysis only examines a single objective and there is more than one objective to realize utility computing, it is critical to be able to assess a combination of multiple objectives in an integrated fashion.

Given that there is a total of o objectives to examine for a particular scenario, the performance μ_{int} and volatility σ_{int} of the integrated risk analysis can be computed using the performance $\mu_{sep,i}$ and volatility $\sigma_{sep,i}$ measures from the separate risk analysis of each objective i:

$$performance, \mu_{int} = \sum_{i=1}^{o} w_i * \mu_{sep,i} \tag{6.7}$$

$$volatility, \sigma_{int} = \sum_{i=1}^{o} w_i * \sigma_{sep,i} \tag{6.8}$$

where $0 \leq w_i \leq 1$ and $\sum_{i=1}^{o} w_i = 1$. w_i is a weight to denote the importance of an objective

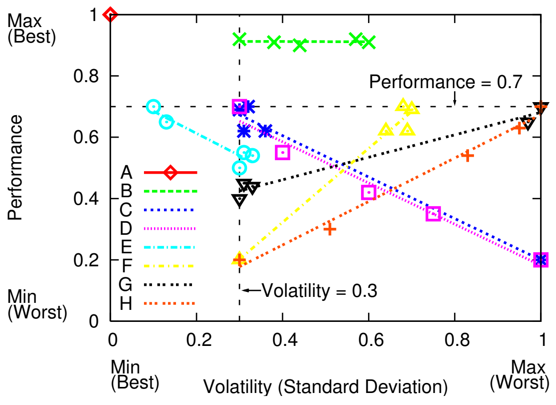


Figure 6.1: Sample risk analysis plot of policies.

i with respect to other objectives. These weights for various objectives provide a flexible means for the service provider to easily adjust the importance of an objective and determine its level of impact on the overall achievement of a combination of objectives. For the experiments, since we consider all the objectives to be of equal importance, w_i of each objective i in a combination of three objectives and all four objectives are thus 0.33 (1/3) and 0.25 (1/4) respectively.

6.3.3 Risk Analysis Plot

Risk analysis plots can now be generated using the performance and volatility values of various resource management policies in all the scenarios. A risk analysis plot can be generated for a single objective (using separate risk analysis) or a combination of objectives (using integrated risk analysis) to easily visualize the level of associated risk for achieving them. Figure 6.1 shows a sample risk analysis plot of eight policies for five scenarios that has been synthetically generated to illustrate how a risk analysis plot should be interpreted.

In a risk analysis plot, each point for a policy represents the performance and volatility of that policy for a particular scenario. Since the sample plot in Figure 6.1 considers five scenarios, there can only be a maximum of five different points for a policy in the plot. A trend line may then be plotted using these points to reflect the general performance and volatility of that policy. A policy cannot have a trend line if it does not have any or too few different points for all the various scenarios. For example, in Figure 6.1, policy A has the same points for all scenarios and thus does not have a trend line.

Table 6.2 shows the maximum and minimum values of the eight policies for performance and volatility in Figure 6.1 and their respective differences. A policy achieving a higher performance is better than achieving a lower performance, whereas a policy achieving a higher volatility is worse than achieving a lower volatility. This is because a higher volatility means that performance results fluctuate more, thus increasing the possibility that the same performance cannot be achieved for various scenarios. In a risk analysis plot, the best performance that can be achieved by a policy is

Table 6.2: Performance and volatility of policies in the sample risk analysis plot.

Policy	**Performance**			**Volatility**		
	Maximum	**Minimum**	**Difference**	**Maximum**	**Minimum**	**Difference**
A	1.0	1.0	0.0	0.0	0.0	0.0
B	0.9	0.9	0.0	0.6	0.3	0.3
C	0.7	0.2	0.5	1.0	0.3	0.7
D	0.7	0.2	0.5	1.0	0.3	0.7
E	0.7	0.5	0.2	0.3	0.1	0.2
F	0.7	0.2	0.5	0.7	0.3	0.4
G	0.7	0.4	0.3	1.0	0.3	0.7
H	0.7	0.2	0.5	1.0	0.3	0.7

Table 6.3: Ranking of policies based on best performance in the sample risk analysis plot.

Rank	**Policy**	**Maximum Performance**	**Minimum Volatility**	**Performance Difference**	**Volatility Difference**	**Gradient of Trend Line**
1	A	1.0	0.0	0.0	0.0	NA
2	B	0.9	0.3	0.0	0.3	zero
3	E	0.7	0.1	0.2	0.2	decreasing
4	G	0.7	0.3	0.3	0.7	increasing
5	F	0.7	0.3	0.5	0.4	increasing
6	C	0.7	0.3	0.5	0.7	decreasing
7	D	0.7	0.3	0.5	0.7	decreasing
8	H	0.7	0.3	0.5	0.7	increasing

the maximum performance value of 1. On the other hand, the best volatility that can be achieved by a policy is the minimum volatility value of 0. Therefore, in Figure 6.1, policy A is the best ideal policy since it achieves the same best ideal performance of 1 for all five scenarios, and thus also the best ideal volatility of 0.

Table 6.3 and 6.4 show the ranking of policies based on best performance and volatility respectively in the sample risk analysis plot. For best performance, a policy is ranked in the following order: (i) maximum performance, (ii) minimum volatility, (iii) performance difference, (iv) volatility difference, and (v) gradient of trend line. For best volatility, the volatility of a policy is first considered before its performance. Hence, for best volatility, a policy is ranked in the following order: (i) minimum volatility, (ii) maximum performance, (iii) volatility difference, (iv) performance difference, and (v) gradient of trend line.

A higher value is preferred for maximum performance, but a lower value is preferred for minimum volatility. For both performance and volatility differences, a lower value is preferred as it represents a shorter range of possibilities. The preferred order of gradient is as follows: (i) de-

Table 6.4: Ranking of policies based on best volatility in the sample risk analysis plot.

Rank	**Policy**	**Minimum Volatility**	**Maximum Performance**	**Volatility Difference**	**Performance Difference**	**Gradient of Trend Line**
1	A	0.0	1.0	0.0	0.0	NA
2	E	0.1	0.7	0.2	0.2	decreasing
3	B	0.3	0.9	0.3	0.0	zero
4	F	0.3	0.7	0.4	0.5	increasing
5	G	0.3	0.7	0.7	0.3	increasing
6	C	0.3	0.7	0.7	0.5	decreasing
7	D	0.3	0.7	0.7	0.5	decreasing
8	H	0.3	0.7	0.7	0.5	increasing

creasing, (ii) increasing, and (iii) zero. A decreasing gradient indicates a lower volatility for higher performance, whereas an increasing gradient indicates a higher volatility. A zero gradient signifies changing volatility with no change in performance. Thus, in Table 6.3 and 6.4, policies C and D are better than policy H as they have decreasing gradients. But, policy C is better than policy D because most of the points (four of five points) for policy C are near to its maximum performance of 0.7 and minimum volatility of 0.3, compared to the evenly distributed points for policy D.

6.4 Performance Evaluation

In order to thoroughly demonstrate the applicability of separate and integrated risk analysis, this section investigates whether a computing service provider is able to achieve the objectives in two possible economic models: (i) commodity market model and (ii) bid-based model. This section first describes the differences between the commodity market model and bid-based model, before specifying the various resource management policies that will be examined for each of them. It then explains the evaluation methodology, followed by outlining the scenarios for the experiments.

6.4.1 Economic Models

In this chapter, there are two differences between the commodity market model and bid-based model. The first difference is how to determine the price. In the commodity market model, the computing service provider specifies the price that users will pay for the amount of resources consumed. Pricing parameters can be usage time and usage quantity, while prices can be flat or variable. A flat price means that pricing is fixed for a certain time period, whereas a variable price means that pricing changes over time. However, the computing service provider can only charge a cost which is lower than or equal to the maximum budget specified by the user when he submits the job. This also means that a job will be rejected by the computing service provider if the expected cost of the job is higher than the specified budget. In the bid-based model, the user provides the bid or price that he will pay the computing service provider for completing the job.

The second difference is any penalty involved when the computing service provider fails to meet a SLA (which is to complete a job within its deadline). In the commodity market model, there is no penalty involved. The computing service provider continues to charge the user based on the usual pricing parameter and price. But in the bid-based model, the computing service provider is liable to be penalized based on the penalty function shown in Figure 4.1. The penalty function penalizes the computing service provider by reducing the budget of a job over time after the lapse of its deadline. For simplicity, we model the penalty function as linear, as in other previous works [45][63][92]. For every job i, the computing service provider earns a utility U_i depending on its penalty rate PR_i and delay DY_i:

$$U_i = B_i - (DY_i * PR_i) \tag{6.9}$$

Assuming that job i has a specified deadline D_i which starts from the time TSU_i when job i is submitted into the computing service, job i will have a delay DY_i if it takes longer to complete than its deadline D_i:

$$DY_i = max(0, (TF_i - TSU_i) - D_i) \tag{6.10}$$

where TF_i is the time when job i is finished. Thus, job i has no delay (i.e. $DY_i = 0$) if it finishes before the deadline and the computing service earns the full budget B_i as utility U_i. But, if there is a delay (i.e. $DY_i > 0$), U_i drops linearly until it turns negative and becomes a penalty (i.e. $U_i < 0$). As shown in Figure 4.1, the penalty is unbounded till the time when the job is finally completed. This implies that the computing service provider must be cautious about accepting new jobs to ensure that not too many jobs are accepted such that heavily penalized jobs dramatically erode previously earned utility.

6.4.2 Resource Management Policies

Table 6.5 lists five resource management policies to be examined for each economic model and the primary scheduling parameter they consider to allocate resources to jobs. We first describe how each policy works in general, before explaining the difference between policies examined in the commodity market model and bid-based model.

FCFS-BF, SJF-BF, and EDF-BF are backfilling policies which prioritize jobs based on arrival time (First Come First Serve), runtime (Shortest Job First), and deadline (Earliest Deadline First) respectively. All three policies adopt EASY backfilling [79][84] to increase resource utilization. A single queue is used to store incoming jobs as only a single job can run on a processor at any time (i.e. space-shared). When insufficient number of processors is available for the first job (with the highest priority) in the queue, EASY backfilling assigns these unused processors to the next waiting

Table 6.5: Policies for performance evaluation.

Policy	**Economic Model**		**Primary Scheduling Parameter**			
	Commodity Market Model	**Bid-based Model**	**Arrival Time**	**Runtime**	**Deadline**	**Budget with Penalty**
FCFS-BF	✓	✓	✠			
SJF-BF	✓			✠		
EDF-BF	✓	✓			✠	
Libra	✓	✓			✠	
Libra+$	✓				✠	
LibraRiskD		✓			✠	
FirstReward		✓				✠

jobs in the queue provided that they do not delay the first job based on their runtime estimates. In other words, jobs that skip ahead must finish before the time when the required number of processors by the first job is expected to be available.

These three variations of EASY backfilling policy are chosen for comparison because EASY backfilling is currently the most widely used policy for scheduling parallel jobs in commercial cluster batch schedulers [51]. However, we find that these policies without job admission control perform much worse, especially when deadlines of jobs are short. Hence, we implement a generous admission control that checks whether a job should be rejected based on two conditions before running it: (i) the job is predicted to exceed its deadline based on its runtime estimate, and (ii) the job has already exceeded its deadline while waiting in the queue. This generous admission control enables FCFS-BF, SJF-BF, and EDF-BF to select their highest priority job at the latest time, while ensuring that earlier jobs whose deadlines have lapsed do not incur propagated delay for later jobs.

Libra [102] uses deadline-based proportional processor share with job admission control to enforce the deadlines of jobs. A minimum processor time share is computed for each job i as RE_i/D_i using its runtime estimate RE_i and deadline D_i so that job i is accepted only if there are sufficient required number of processors with the free minimum processor time share. This means that multiple jobs can run on a processor at any time, using its allocated minimum processor time share (i.e. time-shared). Unlike the above backfilling policies, no queue is maintained so a new job is checked during submission and rejected immediately if its deadline is not expected to be fulfilled. Libra chooses suitable processors based on the best fit strategy, i.e. processors that have the least available processor time left with the new job will be selected first so that every processor is saturated to its maximum. Any remaining free processor time is then distributed among all jobs on the processor according to the computed processor time share of each job.

Libra+$ [128] is Libra with an enhanced pricing function that satisfies four essential requirements for pricing of resources to prevent workload overload: (i) flexible, (ii) fair, (iii) dynamic, and (iv) adaptive. The price P_{ij} for per unit of resource utilized by job i at compute node j is computed as: $P_{ij} = (\alpha * PBase_j) + (\beta * PUtil_{ij})$. The base price $PBase_j$ is a static pricing component for utilizing a resource at node j. The utilization price $PUtil_{ij}$ is a dynamic pricing component which is computed as a factor of $PBase_j$ based on the utilization of the resource at node j for the required deadline of job i: $PUtil_{ij} = RESMax_j / RESFree_{ij} * PBase_j$. $RESMax_j$ and $RESFree_{ij}$ are the maximum units and remaining free units of the resource at node j for the deadline duration of job i respectively. Hence, $RESFree_{ij}$ has been deducted units of resource committed for other confirmed reservations and job i for its deadline duration.

The factors α and β for the static and dynamic components of Libra+$ respectively, provide the flexibility for the service provider to easily configure and modify the weight of the static and dynamic components on the overall price P_{ij}. Libra+$ is fair since jobs are priced based on the amount of different resources utilized. It is also dynamic because the overall price of a job varies depending on the availability of resources for the required deadline. Finally, it is adaptive as the overall price is adjusted depending on the current supply and demand of resources to either encourage or discourage job submission. For the experiments, α is 1 and β is 0.3.

LibraRiskD [127] is also an improvement of Libra and uses the same deadline-based proportional processor share. The difference is that LibraRiskD considers the risk of deadline delay when selecting suitable nodes for a new job. Nodes are selected for a new job only if they have zero risk of deadline delay. This enables LibraRiskD to manage the risk of inaccurate runtime estimates more effectively than Libra. LibraRiskD is thus able to complete more jobs with deadline fulfilled and achieve lower average slowdown than Libra.

FirstReward [63] determines possible future earnings PV_i with possible opportunity cost penalties $cost_i$ based on the penalty rate PR_i and estimated remaining runtime RPT_i of a job i. The reward $reward_i$ is then calculated through a α-weighting function as: $reward_i = ((\alpha * PV_i) - ((1-\alpha) * cost_i))/RPT_i$. The earnings PV_i of a job i is computed as: $PV_i = B_i/(1 + (discount\_rate * RPT_i))$, where B_i is the budget of job i. For unbounded penalties, the penalty cost $cost_i$ of a job i is the sum of penalty for all other n accepted jobs based on RPT_i: $cost_i = \sum_{j=0;j\neq i}^{n}(PR_j * RPT_i)$. The admission control of FirstReward computes the slack $slack_i$ of a new job i during submission and rejects the job immediately if $slack_i$ is less than a specified slack threshold: $slack_i = (PV_i - cost_i)/PR_i$. The slack threshold determines the balance of earnings and penalties where a high threshold avoids future commitments that can result in possible penalties. Setting the correct slack threshold is not trivial as the ideal slack threshold changes depending on the workload. After testing various slack threshold values for the simulated workload, we derive the following ideal simulation settings for FirstReward: α is 1, the discount rate is 1%, and the slack threshold is 25. We have also extended the FirstReward to consider multiple-processor parallel jobs since the original work only considers single-processor jobs. However, we do not make FirstReward to support backfilling, so delays may occur due to waiting for the required number of processors.

Table 6.5 shows that some policies (FCFS-BF, EDF-BF, and Libra) are examined in both commodity market model and bid-based model. As previously explained in Section 6.4.1, the only difference between these two models is how to determine the utility earned by the computing service provider.

In the commodity market model, the policies (FCFS-BF, SJF-BF, EDF-BF, Libra, and Libra+$) earn utility based on the different pricing rates each of them charge. However, the maximum utility that can be earned for a job is restricted by the user-specified budget. In other words, a job that is expected to cost more than the specified budget will be rejected. FCFS-BF, SJF-BF, and EDF-BF charge the user based on the static base price $PBase_j$ of using processing time at node j, so the computing service provider earns a utility of $RE_i * PBase_j$ for job i with runtime estimate RE_i. For the experiments, $PBase_j$ is \$1 per second for all nodes. Libra uses a static pricing function that offers incentives for jobs with a more relaxed deadline to compute a utility of $\gamma * RE_i + \delta * RE_i/D_i$ for job i with runtime estimate RE_i and deadline D_i. γ is a factor for the first component that computes the cost based on the runtime of the job, so that longer jobs are charged more than shorter jobs. δ is a factor for the second component that offers incentives for jobs with a more relaxed deadline, so as to encourage users to submit jobs with longer deadlines. For the experiments, both γ and δ are 1. Libra+$ uses an enhanced pricing function as described earlier in this section. But, for a job i, Libra+$ can compute different price P_{ij} at each node j since workload conditions can vary at different nodes. Hence, to maximize revenue, Libra+$ uses the highest price P_{ij} among allocated nodes as the price for job i.

On the other hand, in the bid-based model, all policies (FCFS-BF, EDF-BF, Libra, LibraRiskD, and FirstReward) can earn a maximum utility equal to the budget (bid) of the job specified by the user for completing the job within its deadline. If these policies cannot complete a job within its deadline, the utility reduces and can instead become a penalty depending on when the job is eventually completed (as described in Section 6.4.1).

Finally, all the policies are assumed to be non-preemptive. In other words, jobs that are started need to complete entirely and are not paused or terminated after the lapse of their deadlines. This leads to the issue of whether the non-preemptive policies will be affected by the inaccuracy of runtime estimates.

Under estimation of runtime estimates of previously accepted jobs can result in delays that cause later accepted jobs not to finish within their expected deadlines. For the commodity market model, potential utility is lost when fewer later arriving jobs are accepted due to the delays caused by previously accepted jobs. For the bid-based model, the loss of utility can be caused by accepting fewer later arriving jobs and paying penalties to compensate users for delays.

Conversely, over estimation of runtime estimates allows fewer jobs to be accepted since admission controls unnecessarily reject jobs after predicting that their deadlines cannot be fulfilled.

For the commodity market model, higher utility may however be gained as the prices charged are computed using the over-estimated runtime estimates. But, for the bid-based model, potential utility is lost as fewer jobs are accepted.

6.4.3 Evaluation Methodology

We use the same evaluation methodology in Section 3.5.1 for the commodity market model and Section 4.4.1 for the bid-based model respectively. But, for both models, the experiments are generated from a subset of the last 5000 jobs in the SDSC SP2 trace (April 1998 to April 2000) version 2.2 from Feitelson's Parallel Workload Archive [2], which is same as in Section 3.5.1. The computing service that is simulated resembles the IBM SP2 at San Diego Supercomputer Center (SDSC) with 128 compute nodes, each having a SPEC rating of 168.

Table 6.6 lists the simulation settings for the experiments. Since the deadline D_i, budget B_i and penalty rate PR_i of a job i will now always be set as a larger factor of the real runtime R_i from the trace, we introduce a *bias* parameter to counter against this issue. For example, the deadline bias BD_i works such that a job i with a runtime more than the average runtime of all jobs (i.e. longer runtime) will have a deadline $D_i = D_i/BD_i$ (i.e. shorter deadline). But if job i has a runtime less than the average runtime of all jobs (i.e. shorter runtime), then it will have $D_i = D_i * BD_i$ (i.e. longer deadline). This works likewise for the budget and penalty bias.

The *inaccuracy* of runtime estimates is measured with respect to the actual runtime estimates of jobs obtained from the trace. An inaccuracy of 0% assumes runtime estimates are accurate and equal to the real runtime of the jobs. On the other hand, an inaccuracy of 100% means that runtime estimates are the actual runtime estimates as reflected in the trace which may be accurate (equal to the real runtime) or inaccurate (not equal to the real runtime). For the actual runtime estimates from the last 5000 job subset of the SDSC SP2 trace, only 8% of them are under estimates, while the remaining 92% of them are over estimates. This means that runtime estimates provided by users are often over estimated.

6.4.4 Scenarios

We can now apply separate and integrated risk analysis (introduced in Section 6.3) to assess the resource management policies with respect to the four essential objectives (defined in Section 6.2). For each objective, we consider twelve scenarios. Table 6.6 lists the twelve scenarios and their varying values for the experiments.

Each of the varying bias, varying high:low ratio, and varying low-value mean scenarios is for the deadline, budget, and penalty parameters respectively, thus creating a total of nine scenarios. For simplicity, we have set the deadline, budget, and penalty parameters to have the same default and varying values for varying bias, varying high:low ratio, and varying low-value mean scenarios. The three other remaining scenarios are varying percentage of high urgency jobs (job mix), varying

Table 6.6: Simulation settings of twelve scenarios.

% of High Urgency Jobs	**Arrival Delay Factor**	**% of Inaccuracy of Runtime Estimates**	**Bias (Deadline, Budget, Penalty)**	**High:low Ratio (Deadline, Budget, Penalty)**	**Low-value Mean (Deadline, Budget, Penalty)**
0	0.02	(Set A) <u>0</u>	<u>1</u>	1	1
<u>20</u>	0.10	20	2	2	2
40	0.25	40	4	<u>4</u>	<u>4</u>
60	0.50	60	6	6	6
80	0.75	80	8	8	8
100	<u>1.00</u>	(Set B) <u>100</u>	10	10	10

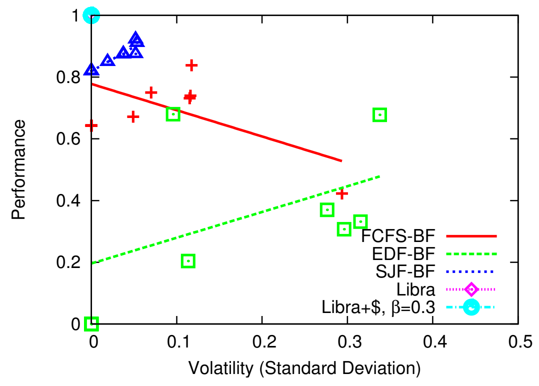
Note: <u>underline</u> denotes default value.

arrival delay factor (workload), and varying percentage of inaccuracy of runtime estimates. For each scenario, there is only one set of varying values, while the rest of the experiment settings remains the same with default values (underlined in Table 6.6). Table 6.6 also shows six varying values in each scenario, thus deriving six normalized results to compute the separate risk analysis of a particular objective.

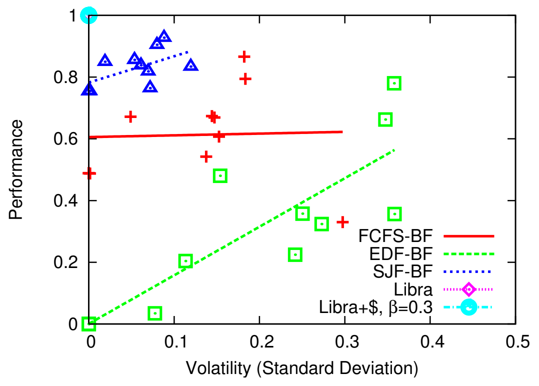
Previous studies [84][113] have shown that runtime estimates provided by users may be rather inaccurate. Hence, for each economic model (commodity market model and bid-based model), we run two different sets of experiments: (i) Set A and (ii) Set B to examine the impact of inaccuracy of runtime estimates on the achievement of objectives. The only different setting between Set A and Set B is the default value for percentage of inaccuracy of runtime estimates as shown in Table 6.6: Set A has 0% of inaccuracy to represent accurate runtime estimates, while Set B has 100% of inaccuracy to represent actual runtime estimates from the trace.

6.5 Performance Results

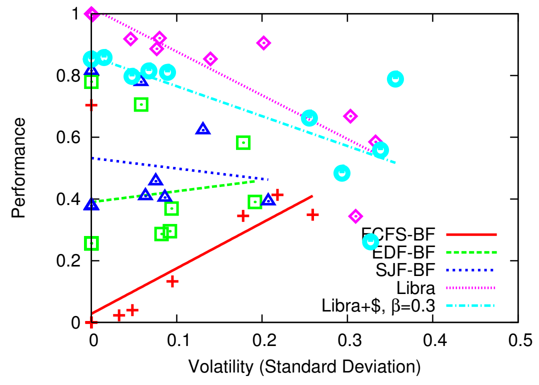
This section analyzes the performance results of various resource management policies for two economic models: (i) commodity market model and (ii) bid-based model. As there are four essential objectives to realize utility computing (as defined in Section 6.2), it examines the performance results using three approaches for each economic model: (i) separate risk analysis of one objective, (ii) integrated risk analysis of three objectives, and (iii) integrated risk analysis of all four objectives. The integrated risk analysis of three objectives enables the understanding of how the combination of all the other remaining objectives will perform in the absence of a particular objective.



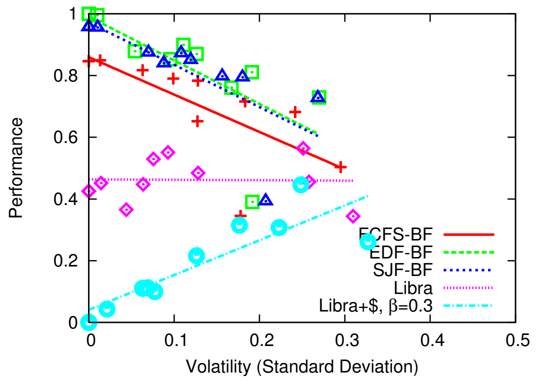
(a) Set A: *wait*



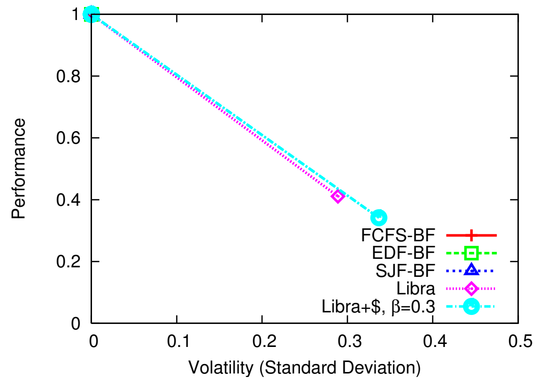
(b) Set B: *wait*



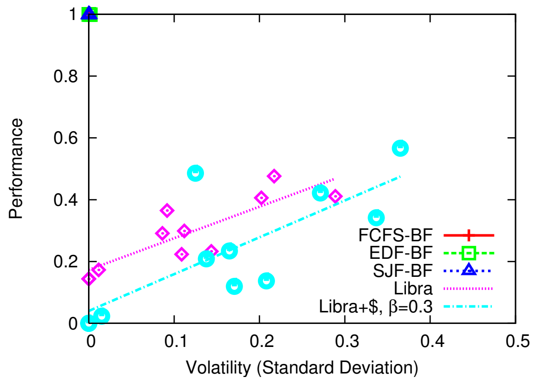
(c) Set A: *SLA*



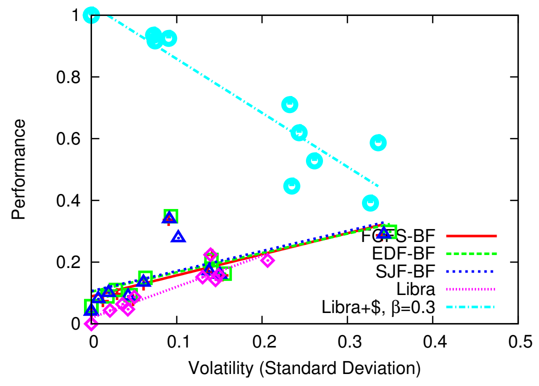
(d) Set B: *SLA*



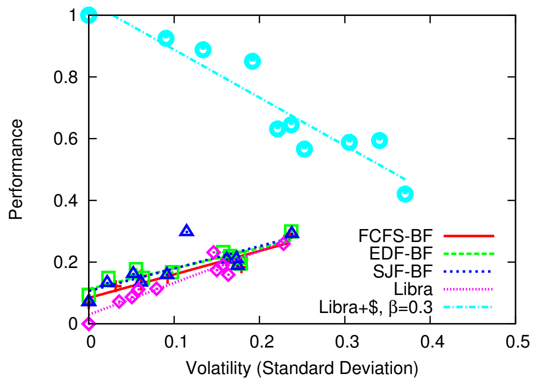
(e) Set A: *reliability*



(f) Set B: *reliability*



(g) Set A: *profitability*



(h) Set B: *profitability*

Figure 6.2: Commodity market model: Separate risk analysis of one objective.

6.5.1 Commodity Market Model

Figure 6.2 shows the separate risk analysis of one objective ($wait$, SLA, $reliability$, and $profitability$) in Set A (accurate runtime estimates) and Set B (actual runtime estimates from the trace) for the commodity market model. For the *wait* objective in Set A (Figure 6.2(a)) and Set B (Figure 6.2(b)), Libra and Libra+\$ have the best ideal performance and volatility of 1 and 0 respectively since jobs are examined immediately after submission to determine whether their deadlines can be fulfilled or not. On the other hand, FCFS-BF, SJF-BF, and EDF-BF have lower performance and higher volatility as they keep jobs in a single queue and examine them only prior to execution to enable a better selection choice. Out of these three policies, SJF-BF returns the best performance and volatility because it selects the shortest job to execute first and thus requires queued jobs to wait the least before being examined. EDF-BF returns the worst performance and volatility since jobs that arrive later but have earlier deadlines will execute first, thus delaying other jobs submitted earlier before them.

For the SLA objective in Set A (Figure 6.2(c)) and Set B (Figure 6.2(d)), SJF-BF has either similar or better performance than EDF-BF, while EDF-BF has better performance than FCFS-BF. This is due to the fact that the evaluation methodology always set the deadline of a job as a larger factor of its runtime for all scenarios (except deadline bias). Thus, SJF-BF has the best performance among these three policies by executing the job with the shortest runtime first, whereas FCFS-BF has the worst performance by considering the arrival time and not deadline.

Libra+\$ has lower performance and slightly higher volatility than Libra in Set A and Set B since it accepts and fulfills a lower number of jobs by increasing the pricing as the workload increases. In Set A, Libra and Libra+\$ have the best performance. But, in Set B, Libra and Libra+\$ have the worst performance for the similar volatility as that of FCFS-BF, SJF-BF, and EDF-BF. In particular, Libra and Libra+\$ have increasing performance with decreasing volatility (decreasing gradient) in Set A which is a better result, compared to constant or increasing performance with increasing volatility (zero and increasing gradient) in Set B which is a worse result.

This highlights the issue of inaccurate runtime estimates. Libra and Libra+\$ assume accurate runtime estimates and thus accept a lower number of jobs in Set B with the actual runtime estimates from the trace being inaccurate. FCFS-BF, SJF-BF, and EDF-BF are less affected than Libra and Libra+\$ in Set B because of the generous admission control we implemented for them. New jobs are only examined and accepted prior to execution when the previously accepted jobs are completed and not during job submission in the case of Libra and Libra+\$.

For the *reliability* objective in Set A (Figure 6.2(e)) and Set B (Figure 6.2(f)), the impact of inaccurate runtime estimates on Libra and Libra+\$ can be clearly seen. In Set A, Libra and Libra+\$ have a single point deviation due to the scenario of inaccuracy of runtime estimates. In Set B, Libra and Libra+\$ have substantially lower performance and higher volatility as the actual runtime estimates from the trace are highly inaccurate. In contrast, FCFS-BF, SJF-BF, and EDF-BF have the best ideal performance and volatility of 1 and 0 respectively in Set A and Set B.

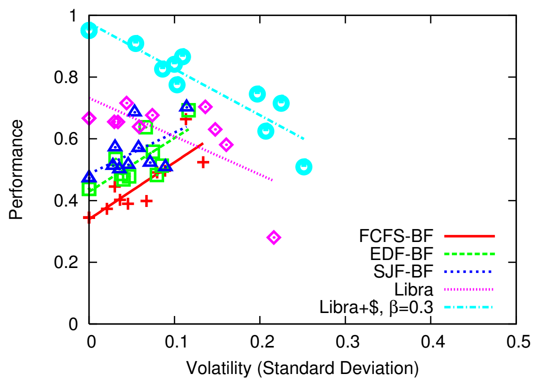
For the *profitability* objective in Set A (Figure 6.2(g)) and Set B (Figure 6.2(h)), Libra+$ has the best performance. In addition, only Libra+$ has increasing performance with decreasing volatility (decreasing gradient), whereas all other policies have increasing performance with increasing volatility (increasing gradient). This demonstrates the effectiveness of the enhanced pricing function used by Libra+$ to gain significantly higher utility than all other policies, even when the number of jobs accepted is lower in Set B (Figure 6.2(d)). However, Libra+$ has higher volatility than all other policies in Set B.

Figure 6.3 shows the integrated risk analysis of three objectives in Set A (accurate runtime estimates) and Set B (actual runtime estimates from the trace) for the commodity market model. For the three combinations of objectives that include the *profitability* objective in Set A (Figure 6.3(a), 6.3(c), and 6.3(e)) and Set B (Figure 6.3(b), 6.3(d), and 6.3(f)), Libra+$ has higher performance than Libra. Libra+$ has lower performance than Libra only for the combination of objectives without the *profitability* objective in Set A (Figure 6.3(g)) and Set B (Figure 6.3(h)). This reinforces that Libra+$ is able to gain higher utility through its enhanced pricing function.

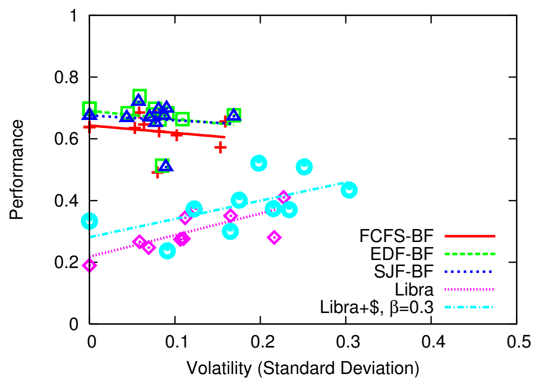
Again, the inaccuracy of runtime estimates can be observed to dramatically affect the performance of Libra and Libra+$. For the combination of objectives without *SLA* (Figure 6.3(d)) and *reliability* (Figure 6.3(f)) objectives, the performance of Libra and Libra+$ are somewhat similar or slightly worse than FCFS-BF, SJF-BF, and EDF-BF. But, for the combination of objectives without *wait* (Figure 6.3(b)) and *profitability* (Figure 6.3(h)) objectives, the performance of Libra and Libra+$ are much worse than FCFS-BF, SJF-BF, and EDF-BF. Libra and Libra+$ are thus able to achieve considerably better performance than FCFS-BF, SJF-BF, and EDF-BF for *wait* and *profitability* objectives.

Another interesting observation pertains to the three backfilling policies: FCFS-BF, SJF-BF, and EDF-BF. For the three combinations of objectives that include the *wait* objective in Set A (Figure 6.3(c), 6.3(e), and 6.3(g)) and Set B (Figure 6.3(d), 6.3(f), and 6.3(h)), SJF-BF has the best performance and volatility, while EDF-BF has the worst performance. When the *wait* objective is excluded in Set A (Figure 6.3(a)) and Set B (Figure 6.3(b)), these three policies have almost similar performance and volatility. This highlights that the amount of wait time for SLA acceptance incurred by these three policies for the *wait* objective critically affects the overall achievement of objectives.

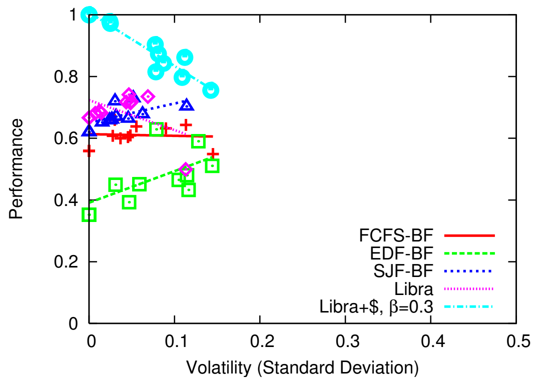
Figure 6.4 shows the integrated risk analysis of all four objectives in Set A (accurate runtime estimates) and Set B (actual runtime estimates from the trace) for the commodity market model. In Set A (Figure 6.4(a)), Libra and Libra+$ have the best performance. In particular, Libra and Libra+$ have increasing performance with decreasing volatility (decreasing gradient) which is better, compared to FCFS-BF, SJF-BF, and EDF-BF which have increasing performance with



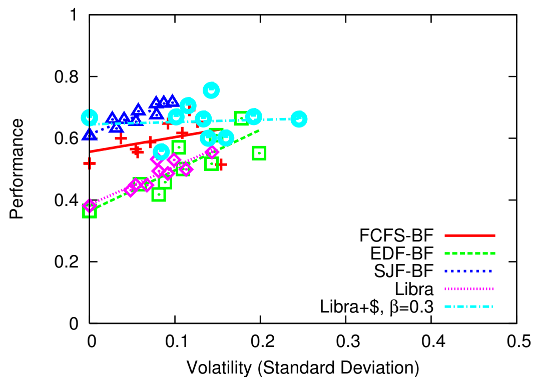
(a) Set A: *SLA*, *reliability*, *profitability*



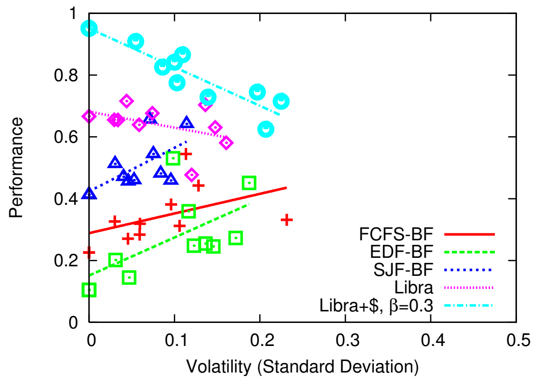
(b) Set B: *SLA*, *reliability*, *profitability*



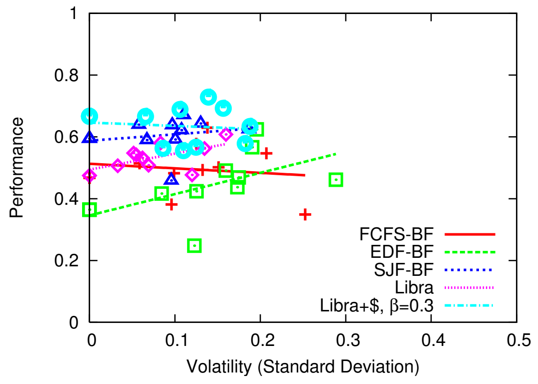
(c) Set A: *wait*, *reliability*, *profitability*



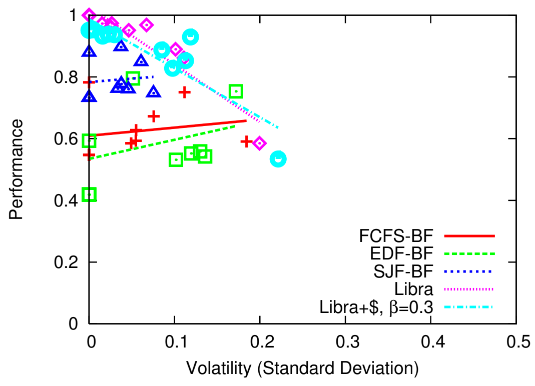
(d) Set B: *wait*, *reliability*, *profitability*



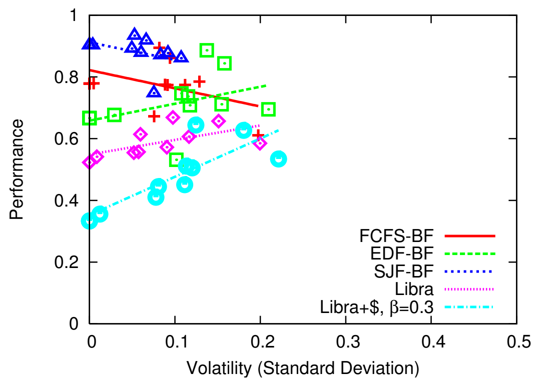
(e) Set A: *wait*, *SLA*, *profitability*



(f) Set B: *wait*, *SLA*, *profitability*

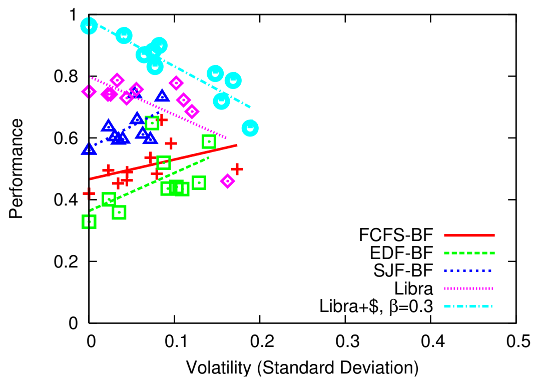


(g) Set A: *wait*, *SLA*, *reliability*

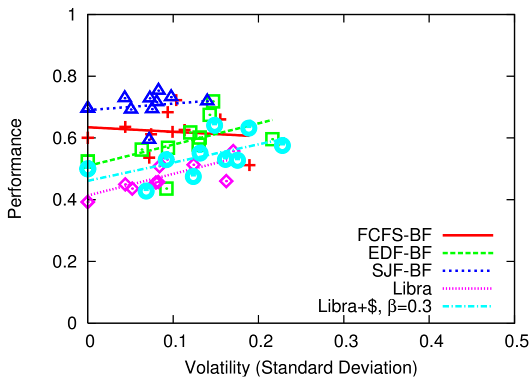


(h) Set B: *wait*, *SLA*, *reliability*

Figure 6.3: Commodity market model: Integrated risk analysis of three objectives.



(a) Set A: *wait*, *SLA*, *reliability*, *profitability*



(b) Set B: *wait*, *SLA*, *reliability*, *profitability*

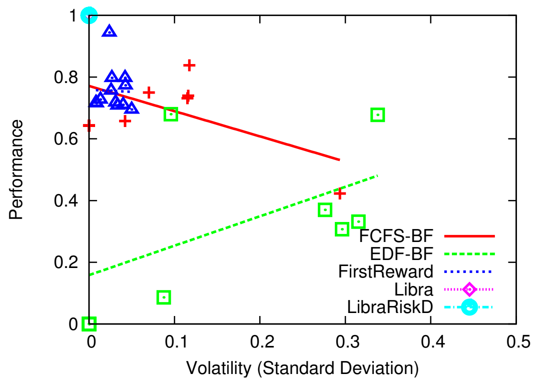
Figure 6.4: Commodity market model: Integrated risk analysis of all four objectives.

increasing volatility (increasing gradient). Libra+$ is able to achieve the best performance due to the capability of its enhanced pricing function to achieve very much better performance for the *profitability* objective.

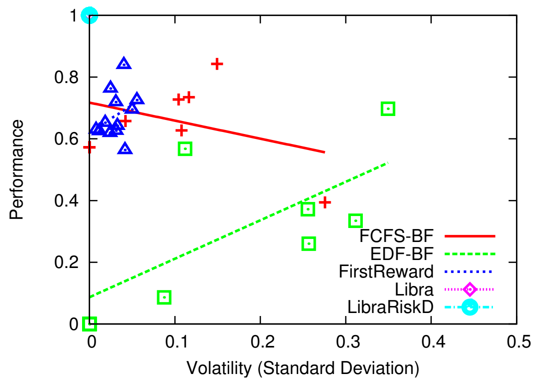
But, in Set B (Figure 6.4(b)), Libra and Libra+$ have the worst performance as the actual runtime estimates from the trace are inaccurate. Libra and Libra+$ also have increasing performance with increasing volatility (increasing gradient) in Set B which is worse, compared to having increasing performance with decreasing volatility (decreasing gradient) in Set A. Instead, SJF-BF has the best performance and volatility in Set B. This exposes the weakness of non-preemptive policies using admission controls that rely on accurate runtime estimates to examine and accept jobs at job submission, especially when the actual runtime estimates from the trace are highly inaccurate.

6.5.2 Bid-based Model

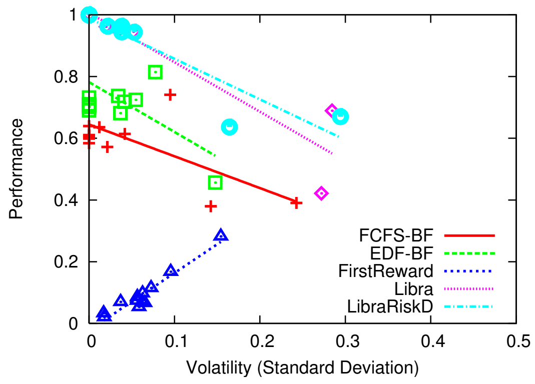
Figure 6.5 shows the separate risk analysis of one objective (*wait*, *SLA*, *reliability*, and *profitability*) in Set A (accurate runtime estimates) and Set B (actual runtime estimates from the trace) for the bid-based model. For the *wait* objective in Set A (Figure 6.5(a)) and Set B (Figure 6.5(b)), Libra and LibraRiskD have the best ideal performance and volatility of 1 and 0 respectively since jobs are examined immediately after submission to determine whether their deadlines can be fulfilled or not. FirstReward has the next best performance and volatility as it also examines new jobs immediately after submission. But, FirstReward has lower performance and higher volatility than Libra and LibraRiskD because of two reasons. The first reason is that FirstReward delays previously accepted jobs to accept new jobs that are more profitable. The second reason is that FirstReward is unable to start the execution of accepted jobs immediately if the required number of processors is not available due to its assumption of space-shared execution. On the other hand, Libra and LibraRiskD assume time-shared execution and immediately starts the execution of accepted jobs by allocating processor time based on the deadline and runtime estimate of each job.



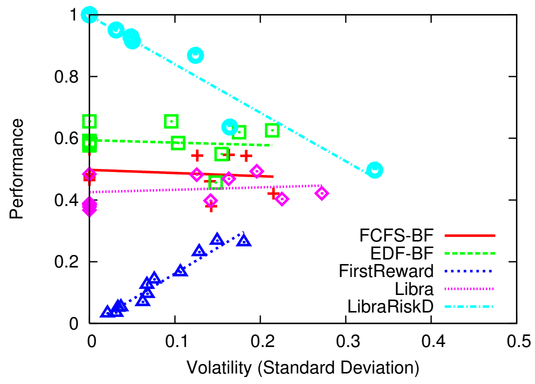
(a) Set A: *wait*



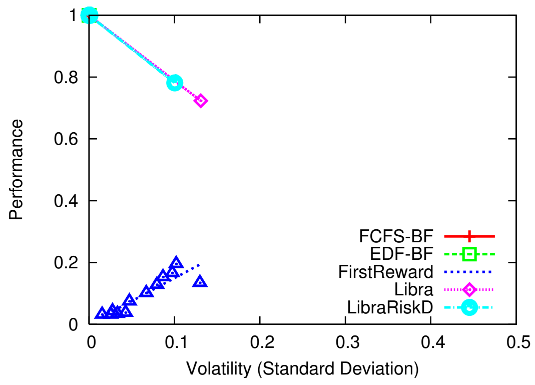
(b) Set B: *wait*



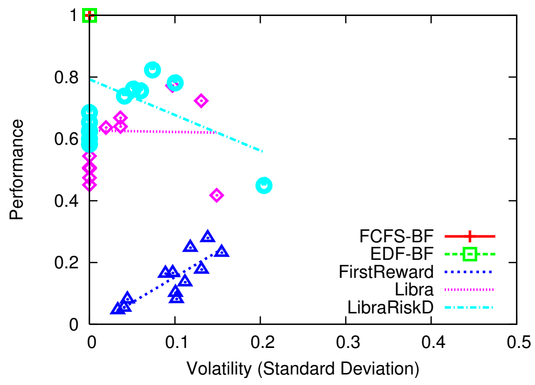
(c) Set A: *SLA*



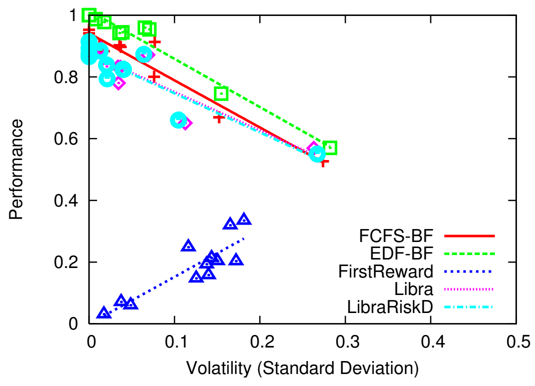
(d) Set B: *SLA*



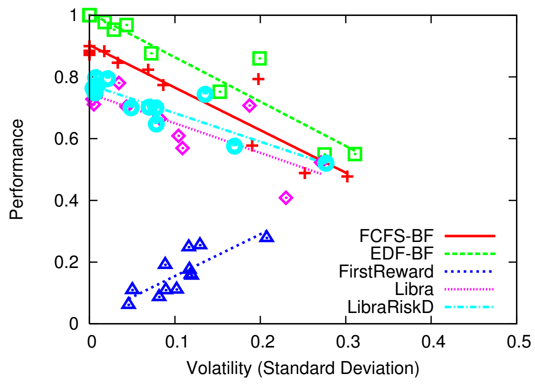
(e) Set A: *reliability*



(f) Set B: *reliability*



(g) Set A: *profitability*



(h) Set B: *profitability*

Figure 6.5: Bid-based model: Separate risk analysis of one objective.

For the *SLA* objective in Set A (Figure 6.5(c)) and Set B (Figure 6.5(d)), FirstReward has the worst performance, but the best volatility as it accepts fewer jobs than the other policies. This is because FirstReward is more risk-averse when considering unbounded penalty and thus accepts fewer jobs to reduce the possibility of incurring penalty. Another possible reason is that FirstReward does not support backfilling and thus may accept fewer jobs compared to FCFS-BF and EDF-BF.

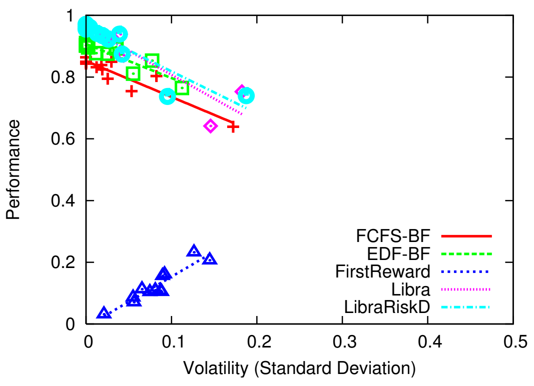
FCFS-BF, EDF-BF, and Libra have constant performance with increasing volatility (zero gradient) in Set B which is worse, compared to having increasing performance with decreasing volatility (decreasing gradient) in Set A. However, LibraRiskD is able to maintain increasing performance with decreasing volatility (decreasing gradient) to record the best performance in Set A and Set B. Although LibraRiskD also has the worst volatility among all the policies, the maximum volatility is only caused by a single point deviation and therefore only applies for one or a few scenarios. It is clear that the main concentration of points for LibraRiskD is close to the best ideal performance and volatility of 1 and 0 respectively. This thus shows that LibraRiskD is able to manage the inaccuracy of runtime estimates a lot better than the other policies.

For the *reliability* objective in Set A (Figure 6.5(e)) and Set B (Figure 6.5(f)), FCFS-BF and EDF-BF has the best ideal performance and volatility of 1 and 0 respectively due to their generous admission control examining and accepting new jobs only after the previously accepted jobs are completed. FirstReward has the worst performance in Set A and Set B because it delays previously accepted jobs to accommodate new jobs if the new jobs can still return higher utility after taken into consideration the penalties incurred for delaying the previously accepted jobs. But, FirstReward has slightly higher performance and volatility in Set B than in Set A.

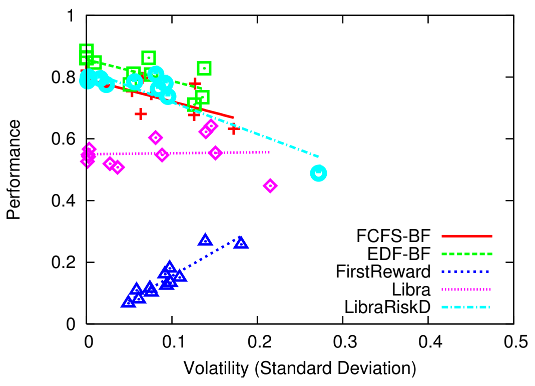
In Set A, Libra and LibraRiskD have a single point deviation due to the scenario of inaccuracy of runtime estimates. In Set B, LibraRiskD achieves higher performance and volatility than Libra, but the higher volatility is only through a single point deviation. Moreover, LibraRiskD has increasing performance with decreasing volatility (decreasing gradient), whereas Libra has constant performance with increasing volatility (zero gradient). Therefore, LibraRiskD is able to manage the inaccuracy of runtime estimates better than Libra.

For the $profitability$ objective in Set A (Figure 6.5(g)) and Set B (Figure 6.5(h)), FirstReward has the worst performance, but the best volatility. In addition, FirstReward has increasing performance with increasing volatility (increasing gradient) which is worse than all the other policies which have increasing performance with decreasing volatility (decreasing gradient). This is due to FirstReward being more risk-averse by accepting fewer jobs and not supporting backfilling. All the other policies also have similar volatility. EDF-BF and FCFS-BF have the best performance, followed by Libra and LibraRiskD, but in Set B, LibraRiskD has a marginally higher performance than Libra.

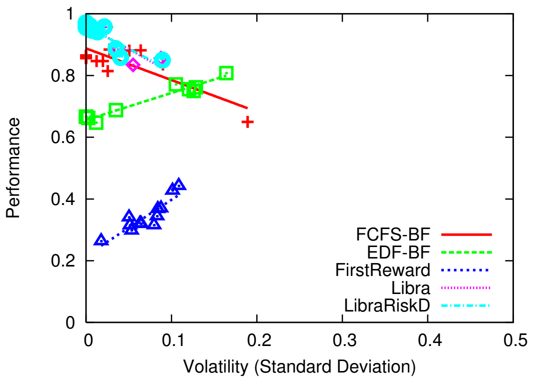
Figure 6.6 shows the integrated risk analysis of three objectives in Set A (accurate runtime



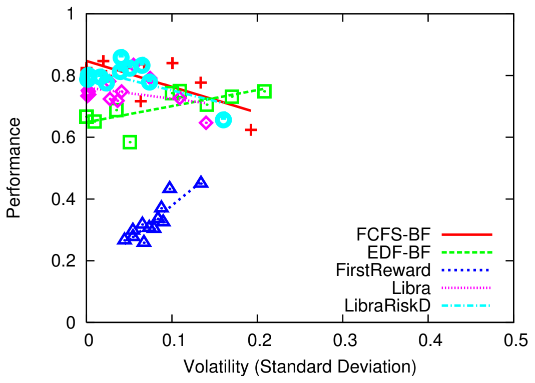
(a) Set A: *SLA*, *reliability*, *profitability*



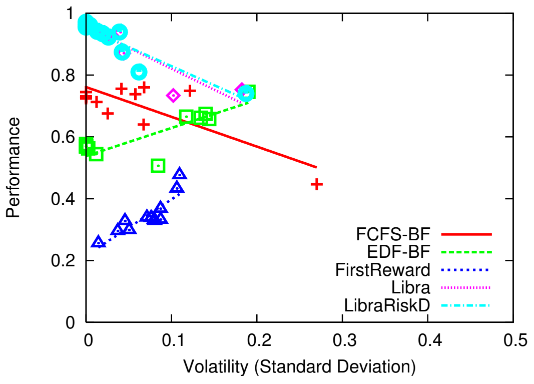
(b) Set B: *SLA*, *reliability*, *profitability*



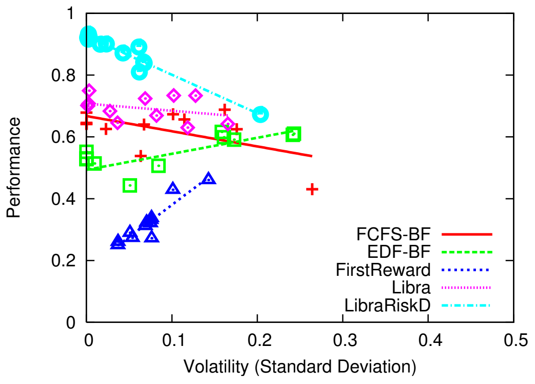
(c) Set A: *wait*, *reliability*, *profitability*



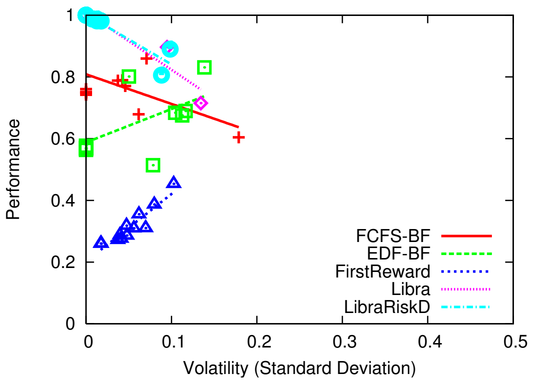
(d) Set B: *wait*, *reliability*, *profitability*



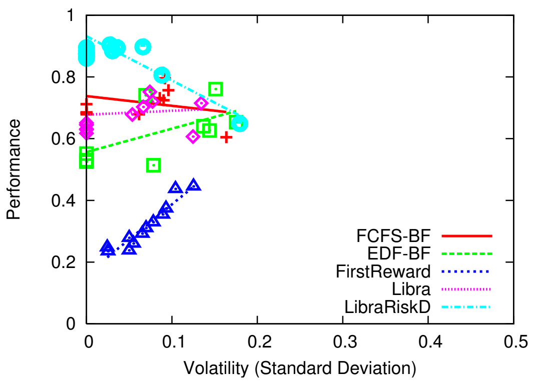
(e) Set A: *wait*, *SLA*, *profitability*



(f) Set B: *wait*, *SLA*, *profitability*

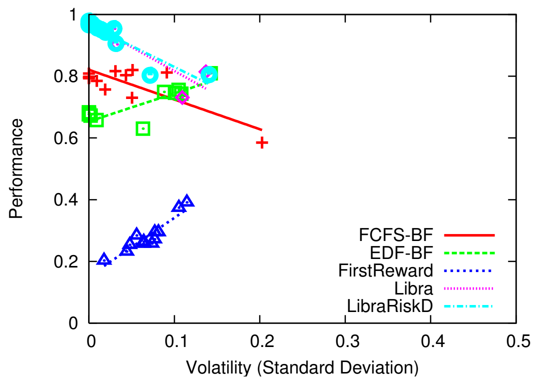


(g) Set A: *wait*, *SLA*, *reliability*

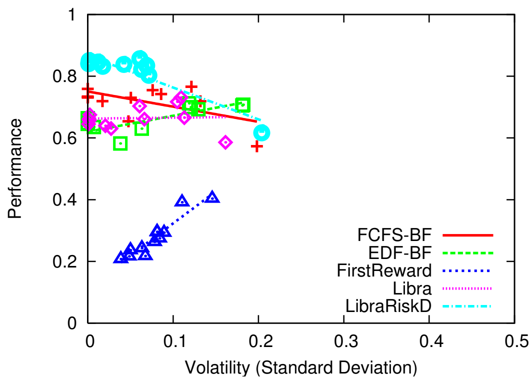


(h) Set B: *wait*, *SLA*, *reliability*

Figure 6.6: Bid-based model: Integrated risk analysis of three objectives.



(a) Set A: $wait$, SLA, $reliability$, $profitability$



(b) Set B: $wait$, SLA, $reliability$, $profitability$

Figure 6.7: Bid-based model: Integrated risk analysis of all four objectives.

estimates) and Set B (actual runtime estimates from the trace) for the bid-based model. For the four possible combinations of objectives in Set A (Figure 6.6(a), 6.6(c), 6.6(e), and 6.6(g)), LibraRiskD has similar performance and volatility as Libra. For the three combinations of objectives that include the SLA objective in Set B (Figure 6.6(b), 6.6(f), and 6.6(h)), LibraRiskD has considerably better performance, but slightly worse volatility than Libra. This reflects that LibraRiskD is able to perform better than Libra through the SLA objective when the runtime estimates are inaccurate.

FirstReward has the worst performance, but the best volatility for the four possible combinations of objectives in Set A and Set B. For the combinations of objectives without $wait$ (Figure 6.6(a) and 6.6(b)) and SLA (Figure 6.6(c) and 6.6(d)) objectives, the difference in performance between FirstReward and the other policies is much greater, as compared to the combinations of objectives without $reliability$ (Figure 6.6(e) and 6.6(f)) and $profitability$ (Figure 6.6(g) and 6.6(h)) objectives. This means that FirstReward performs worse than the other policies due to $wait$ and SLA objectives.

Figure 6.7 shows the integrated risk analysis of all four objectives in Set A (accurate runtime estimates) and Set B (actual runtime estimates from the trace) for the bid-based model. In Set A (Figure 6.7(a)), Libra and LibraRiskD have the same best performance which is higher than FCFS-BF, EDF-BF, and FirstReward. They also have slightly worse volatility than FirstReward which has the best volatility. But, Libra and LibraRiskD are better than FirstReward since the points of Libra and LibraRiskD are concentrated around the best ideal performance and volatility of 1 and 0 respectively, while the points of FirstReward are evenly spread. In addition, Libra and LibraRiskD have increasing performance with decreasing volatility (decreasing gradient), whereas FirstReward has increasing performance with increasing volatility (increasing gradient).

However, in Set B (Figure 6.7(b)), Libra has worse performance than FCFS-BF and EDF-BF, as compared to LibraRiskD which is still able to maintain the best performance. Libra is thus greatly affected by the inaccuracy of runtime estimates, while LibraRiskD is able to manage the inaccuracy of runtime estimates. LibraRiskD also has more points closer to the best ideal performance and volatility of 1 and 0 respectively than the other policies.

6.6 Summary

This chapter describes four essential objectives that need to be considered by a computing service provider in order to realize utility computing: (i) manage wait time for SLA acceptance, (ii) meet SLA requests, (iii) ensure reliability of accepted SLA, and (iv) attain profitability. Two evaluation methods called separate and integrated risk analysis are then proposed to examine whether resource management policies are able to achieve the objectives.

Simulation results have demonstrated that both separate and integrated risk analysis enables the detailed study of various policies with respect to the achievement of a single objective and a combination of objectives respectively. In particular, an objective that is not achieved can severely impact on the overall achievement of other objectives. It is thus essential to examine the achievement of all key objectives together, rather than each standalone objective to correctly identify the best policy that can meet all the objectives. As such, the following summary about the various policies focuses on the achievement of all four objectives together (i.e. integrated risk analysis of all four objectives). Simulation results also reveal that the inaccuracy of actual runtime estimates from the trace can adversely affect the achievement of objectives by non-preemptive policies. In the SDSC SP2 trace used in the simulation, only 8% of the runtime estimates are under estimates, while the remaining 92% are over estimates.

For the commodity market model, Libra+$ is the best policy when runtime estimates are assumed to be accurate. Libra+$ returns significantly higher utility for the computing service provider even though fewer jobs are accepted by increasing its pricing as workload increases. But, Libra+$ performs worse than FCFS-BF, SJF-BF, and EDF-BF when the actual runtime estimates from the trace is highly inaccurate. For the bid-based model, LibraRiskD is the best policy as it can achieve the best performance even when the runtime estimates from the trace are inaccurate. This is because LibraRiskD not only accepts more jobs given that runtime estimates are often over estimated, but also considers the risk of deadline delay for runtime estimates that are under estimated. Thus, Libra+$ and LibraRiskD are shown to be more effective than Libra for the commodity market model and bid-based model respectively.

FCFS-BF, SJF-BF, and EDF-BF are also less affected by the inaccuracy of runtime estimates as they keep submitted jobs in a single queue and accept them only prior to execution. Another advantage of this approach is the better selection choice from more jobs in the queue. However, the tradeoff is the longer wait time for SLA acceptance of jobs that lowers the performance of the *wait* objective. There is also minimal impact on FirstReward since it is already more risk-averse when considering unbounded penalty and thus accepts fewer jobs to reduce the possibility of incurring penalty. But, FirstReward has the worst performance if the runtime estimates are accurate.

Chapter 7

Autonomic Metered Pricing and its Implementation in Enterprise Grids

An increasing number of providers are offering utility computing services which require users to pay only when they use. Most of these providers currently charge users for metered usage based on fixed prices. But, we advocate the use of variable prices by providers to maximize revenue. In this chapter, we highlight the significance of deploying an autonomic pricing mechanism that self-adjusts pricing parameters to consider both application and service requirements of users. Performance results observed in the actual implementation of an enterprise Grid show that the autonomic pricing mechanism is able to achieve higher revenue than various other common pricing mechanisms.

7.1 Related Work

An increasing number of providers are starting to offer utility computing services using metered pricing. But currently, these providers follow a fairly simple pricing scheme to charge users – *fixed* prices based on various resource types as listed in Table 7.1. For processing power, Amazon [11] charges \$0.10 per virtual computer instance per hour (Hr), Sun Microsystems [109] charges \$1.00 per processor (CPU) per Hr, and Tsunamic Technologies [114] charges \$0.85 per CPU per Hr.

This paper focuses on the issue of how to charge commercial users or enterprises which make heavy demands on computing resources, as compared to personal users or individuals with considerably lower requirements. Instead of charging fixed prices for these heavy users, we advocate charging *variable prices* through the use of advanced reservations. Advance reservations are bookings made in advance to secure an available item in the future and are used in the airline, car rental, and hotel industries. In the context of utility computing, an advance reservation is a guarantee of access to a computing resource at a particular time in the future for a particular duration [55].

Charging fixed prices in utility computing is not fair to both the provider and users since

Table 7.1: Fixed prices charged by selected providers (as at 01 November 2007).

Provider (Product)	**Resource Type**							
	Processing Power		**Memory**		**Storage**		**Bandwidth**	
	Speed	**Price**	**Size**	**Price**	**Size**	**Price**	**Speed**	**Price**
Amazon (EC2) [11]	1.7Ghz/↯ Max: 20↯	\$0.10/↯/Hr	1.75GB/↯ Max: 20↯	free	160GB/↯ Max: 20↯	free	250Mb/s	UL: \$0.10/GB DL: \$0.18/GB
Sun (Sun Grid) [109]	unknown	\$1.00/CPU/Hr	4GB/CPU	free	10GB	free	300Mb/s	free
Tsunamic (COD) [114]	unknown	\$0.85/CPU/Hr	4GB/CPU	free	10GB	free	unknown	free

Legend	↯: Virtual computer instance	Hr: Hour	UL: Upload
	CPU: Processor	Max: Maximum quota	DL: Download

different users have distinctive needs and demand specific QoS for various resource requests that can change anytime. In economics, a seller with constrained capacity can adjust prices to maximize revenue if the following four conditions are satisfied [90]: (i) demand is variable but follows a predictable pattern, (ii) capacity is fixed, (iii) inventory is perishable (wasted if unused), and (iv) seller has the ability to adjust prices. Thus, for utility computing, providers can charge variable prices since: (i) demand for computing resources changes but can be expected using advanced reservations [55], (ii) only a limited amount of resources is available at a particular site owned by a provider, (iii) processing power is wasted if unused, and (iv) a provider can change prices.

The main aim of providers charging variable prices is to maximize revenue by differentiating the value of computing services provided to different users. Since providers are commercial businesses driven by profit, they need to maximize revenue. Profitable providers can then fund further expansions and enhancements to improve their utility computing service offerings. Charging variable prices is also particularly useful for resource management as it can result in the diversion of demand from high-demand time periods to low-demand time periods [90], thus maximizing utilization for a utility computing service. Higher prices increase revenue as users who need services during high-demand time periods are willing to pay more, whereas others will shift to using services during low-demand periods. The latter results in higher utilization during these otherwise underutilized low-demand periods and hence lead to higher revenue.

Many market-based RMSs have been implemented across numerous computing platforms [126] that include clusters, distributed databases, Grids, parallel and distributed systems, peer-to-peer, and World Wide Web. To manage resources, these systems adopt a variety of economic models [28], such as auction, bargaining, bartering, commodity market, bid-based proportional resource sharing, posted price, and tendering/contract-net. In this chapter, we examine metered pricing which is applicable in commodity market and posted price models.

Recently, several works have discussed pricing for utility or on-demand computing services. Price-At-Risk [87] considers uncertainty in the pricing decision for utility computing services which have uncertain demand, high development costs, and short life cycle. Pricing models for on-demand computing [61] have been proposed based on various aspects of corporate computing infrastructure which include cost of maintaining infrastructure in-house, business value of infrastructure, scale of infrastructure, and variable costs of maintenance. Another author [48] considers economic aspects of a utility computing service whereby high prices denote higher service level for faster computation. But, these works do not consider autonomic pricing that addresses users' application requirements such as parallel applications and service requirements such as deadline and budget.

Setting variable prices is known as *price discrimination* in economics [88]. There can be three types of price discrimination: (i) first degree, (ii) second degree, and (iii) third degree. In first degree price discrimination, a buyer pays the maximum price that he is willing to pay, therefore price varies by buyer. In second degree price discrimination, a buyer pays a lower price for buying a larger amount, thus price varies by quantity purchased. In third degree price discrimination, a buyer pays a price based on the type of market segment he is from, hence price varies by market segments.

Sulistio et al. [107] have examined third degree price discrimination by using revenue management [90] to determine the pricing of advanced reservations in Grids. It evaluates revenue performance across multiple Grids for variable pricing based on the combination of three market segments of users (premium, business, and budget) and three time periods of resource usage (peak, off-peak, and saver). Hence, it does not derive fine-grained variable prices that differentiate specific application and service requirements of individual users.

Chen et al. [39] have proposed pricing-based strategies for autonomic control of web servers. It uses pricing and admission control mechanisms to control QoS of web requests such as slowdown and fairness. However, we focus on high-performance applications and user-centric service requirements (deadline and budget).

We demonstrate the feasibility of an autonomic pricing mechanism which is extended from Libra+$ [128] using actual implementation in an enterprise Grid. The autonomic version of Libra+$ is able to automatically adjust pricing parameters and hence does not rely on static pricing parameters to be configured manually by the provider in the case of Libra+$.

7.2 Economic Aspects of a Utility Computing Service

This section examines various economic aspects of a utility computing service including: (i) economic models for resource management, (ii) variable pricing with advanced reservations, (iii) pricing issues, and (iv) pricing mechanisms. We consider a scenario wherein a provider owns a set of resources that we term as compute nodes. Each node can be subdivided into resource partitions and leased to users for certain time intervals.

7.2.1 Economic Models for Resource Management

Metered pricing is based on the *commodity market* model [28] whereby users pay providers based on the amount of resources they consume. The prices to be paid by users are primarily determined by providers. Thus, the commodity market model is different from other economic models that require users (not providers) to specify prices, such as auction and bid-based proportional resource sharing, or partially determine prices through negotiation, such as bargaining and tendering/contract-net.

Currently, most providers employ the *posted price* model [28] which is similar to the commodity market model, but prices are openly advertised so that users are aware of the prices they are expected to pay. Table 7.1 lists providers which advertise fixed prices for utilizing various resource types. The posted price model is ideal for providers to initially promote utility computing services to users. Advertising the prices to be charged helps users to compare and decide whether the computing service offered by a particular provider is competitive and suitable to them. The posted price model can thus motivate users to consider cheaper special offers and attract new users to establish market share.

7.2.2 Variable Pricing with Advanced Reservations

The use of advanced reservations [55] has been proposed to provide QoS guarantees for accessing various resources across independently administered systems such as Grids. With advanced reservations, users are able to secure resources required in the future which is important to ensure successful completion of time-critical applications such as real-time and workflow applications or parallel applications requiring a number of processors to run. The provider is able to predict future demand and usage more accurately. Using this knowledge, the provider can apply revenue management [90] to determine pricing at various times to maximize revenue. Once these prices are advertised, users are able to decide in advance where to book resources according to their requirements and their resulting expenses.

Commercial users or enterprises are now able to secure resources in advance with known variable prices. This reinforces the suitability of offering variable prices since enterprises are still able to know the expected prices they need to pay, similar to fixed prices. Enterprises also gain the assurance that they will not be caught unaware or disadvantaged by having to pay unexpected variable prices after utilizing the resources as opposed to known fixed prices.

Having prior knowledge of expected costs is highly critical for enterprises to successfully plan and manage their operations. Resource supply guarantee also allows enterprises to contemplate and target future expansion more confidently and accurately. Enterprises are thus able to scale their reservations accordingly based on short-term, medium-term, and long-term commitments.

Users may face the difficulty of choosing the best price for reserving resources from different utility computing services at different times. This difficulty can be overcome by using resource brokers [117] which act on the behalf of users to identify suitable utility computing services and compare their prices.

7.2.3 Pricing Issues

For simplicity, we examine metered pricing within a utility computing service with constrained capacity and do not consider external influences that can be controlled by the provider, such as cooperating with other providers to increase the supply of resources [75] or competing with them to increase market share [85]. Price protection and taxation regulations from authorities and inflation are beyond the control of the provider.

We assume that users have to pay in order to guarantee reservations. Thus, a utility computing service requires full payment from users at time of reservation to reserve computing resources in advance. We also assume that reservations cannot be cancelled after confirmation at time of reservation. Hence, users are not entitled to any refunds if they do not utilize any of their reserved resources later. Users are also unable to change confirmed reservations as making a change requires the cancellation of the old reservation and booking of the new reservation. Enforcing upfront payments with no cancellation discourages malicious users from making fake reservation requests to increase prices and deprive genuine users of affordable prices. It also prevents revenue loss for the provider when users cancel their advance reservations.

We assume that the execution time period of applications will be less than the reservation time period. In order to enforce other scheduled reservations, a utility computing service will terminate any outstanding applications that are still executing once the time period of reservation expires. This implies that users must ensure that time periods of reservations are sufficient for their applications to be completed. Users are also personally responsible for ensuring that their applications can fully utilize the reserved resources. It is thus disadvantageous to the users if their applications fail to use the entire amount of reserved resources that they have already paid for.

7.2.4 Pricing Mechanisms

We compare three types of pricing mechanisms: (i) Fixed, (ii) FixedTime, and (iii) Libra+$. As listed in Table 7.2, each pricing mechanism has maximum and minimum types which are configured accordingly to highlight the performance range of the pricing mechanism. Fixed charges a fixed price for per unit of resource partition at all times. FixedTime charges a fixed price for per unit of resource partition at different time periods of resource usage where a lower price is charged for off-peak (12AM–12PM) and a higher price for peak (12PM–12AM) .

Libra+$ [128] uses a more fine-grained pricing function that satisfies four essential requirements

Table 7.2: Pricing for processing power.

Name	**Configured Pricing Parameters**
FixedMax	\$3/CPU/Hr
FixedMin	\$1/CPU/Hr
FixedTimeMax	\$1/CPU/Hr (12AM–12PM) \$3/CPU/Hr (12PM–12AM)
FixedTimeMin	\$1/CPU/Hr (12AM–12PM) \$2/CPU/Hr (12PM–12AM)
Libra+\$Max	\$1/CPU/Hr ($PBase_j$), $\alpha = 1$, $\beta = 3$
Libra+\$Min	\$1/CPU/Hr ($PBase_j$), $\alpha = 1$, $\beta = 1$
Libra+\$Auto	same as Libra+\$Min

for pricing of resources to prevent workload overload: (i) flexible, (ii) fair, (iii) dynamic, and (iv) adaptive. The price P_{ij} for per unit of resource partition utilized by reservation request i at compute node j is computed as: $P_{ij} = (\alpha{*}PBase_j)+(\beta{*}PUtil_{ij})$. The base price $PBase_j$ is a static pricing component for utilizing a resource partition at node j which can be used by the provider to charge the minimum price so as to recover the operational cost. The utilization price $PUtil_{ij}$ is a dynamic pricing component which is computed as a factor of $PBase_j$ based on the availability of the resource partition at node j for the required deadline of request i: $PUtil_{ij} = RESMax_j/RESFree_{ij} * PBase_j$. $RESMax_j$ and $RESFree_{ij}$ are the maximum units and remaining free units of the resource partition at node j for the deadline duration of request i respectively. Thus, $RESFree_{ij}$ has been deducted units of resource partition committed for other confirmed reservations and request i for its deadline duration.

The factors α and β for the static and dynamic components of Libra+\$ respectively, provide the flexibility for the provider to easily configure and modify the weight of the static and dynamic components on the overall price P_{ij}. Libra+\$ is fair since requests are priced based on the amount of different resources utilized. It is also dynamic because the overall price of a request varies depending on the availability of resources for the required deadline. Finally, it is adaptive as the overall price is adjusted (depending on the current supply and demand of resources) to either encourage or discourage request submission.

Fixed, FixedTime, and Libra+\$ rely on static pricing parameters that are difficult to be accurately derived by the provider to produce the best performance where necessary. Hence, we propose Libra+\$Auto, an autonomic Libra+\$ that automatically adjusts β based on the availability of compute nodes. Libra+\$Auto thus considers the pricing of resources across nodes, unlike Libra+\$ which only considers pricing of resources at each node j via P_{ij}.

Figure 7.1 shows the pseudocode for adjusting β in Libra+\$Auto. First, the previous dynamic factor $\beta Prev$ is computed as the average of dynamic factors β at n number of allocated compute nodes for the previous reservation request, and the maximum number of nodes is assigned (line 1–2). Then, at each node, the free and reserved number of nodes is determined for the proposed

Figure 7.1: Pseudocode for adjusting β in Libra+$Auto.

```
1  $\beta Prev \leftarrow (\sum_{i=1}^{n} \beta_i$ at compute node $i$) / $n$;
2  $maxNodes \leftarrow$ maximum number of nodes;
3  foreach node $j$ do
4      $freeNodes \leftarrow$ free number of nodes for proposed time slot at node $j$;
5      $reservedNodes \leftarrow maxNodes - freeNodes$ ;
6      if $freeNodes = 0$ then
7          $freeNodes \leftarrow 1$;
8      end
9      $ratioFree \leftarrow maxNodes$ / $freeNodes$ ;
10     if $reservedNodes = 0$ then
11         $reservedNodes \leftarrow 1$;
12     end
13     $ratioReserved \leftarrow$ reservedNodes / $maxNodes$ ;
14     if previous request meets budget then
15         $\beta \leftarrow \beta Prev * ratioFree$ ;
16     else
17         $\beta \leftarrow \beta Prev * ratioReserved$ ;
18     end
19 end
```

time slot at the node (line 3–5). After that, the new dynamic factor β for the node is updated depending on the outcome of the previous request. $\beta Prev$ is increased to accumulate more revenue if the previous request meets the user-defined budget, otherwise it is reduced (line 14–18). For increasing $\beta Prev$, a larger increase is computed when there are less free nodes left for the proposed time slot so as to maximize revenue with decreasing capacity (line 6–9). Conversely, for reducing $\beta Prev$, a larger reduction is computed when there are more free nodes left for the proposed time slot in order not to waste unused capacity (line 10–13).

7.3 System Implementation

This section describes how metered pricing for a utility computing service can be implemented using an enterprise Grid with advanced reservations. An enterprise Grid [40] harnesses unused computing resources of desktop computers (connected over an internal network or the Internet) within an enterprise without affecting the productivity of their users. Hence, it increases the amount of computing resources available within an enterprise to accelerate application performance. Our implementation uses a .NET-based service-oriented enterprise Grid platform called Aneka [42].

7.3.1 Aneka: A Service-oriented Enterprise Grid Platform

Aneka [42] is designed to support multiple application models, persistence and security solutions, and communication protocols such that the preferred selection can be changed at anytime without affecting an existing Aneka ecosystem. To create an enterprise Grid, the provider only needs to start an instance of the configurable Aneka container hosting required services on each selected desktop node. The purpose of the Aneka container is to initialize services and acts as a single point for interaction with the rest of the enterprise Grid.

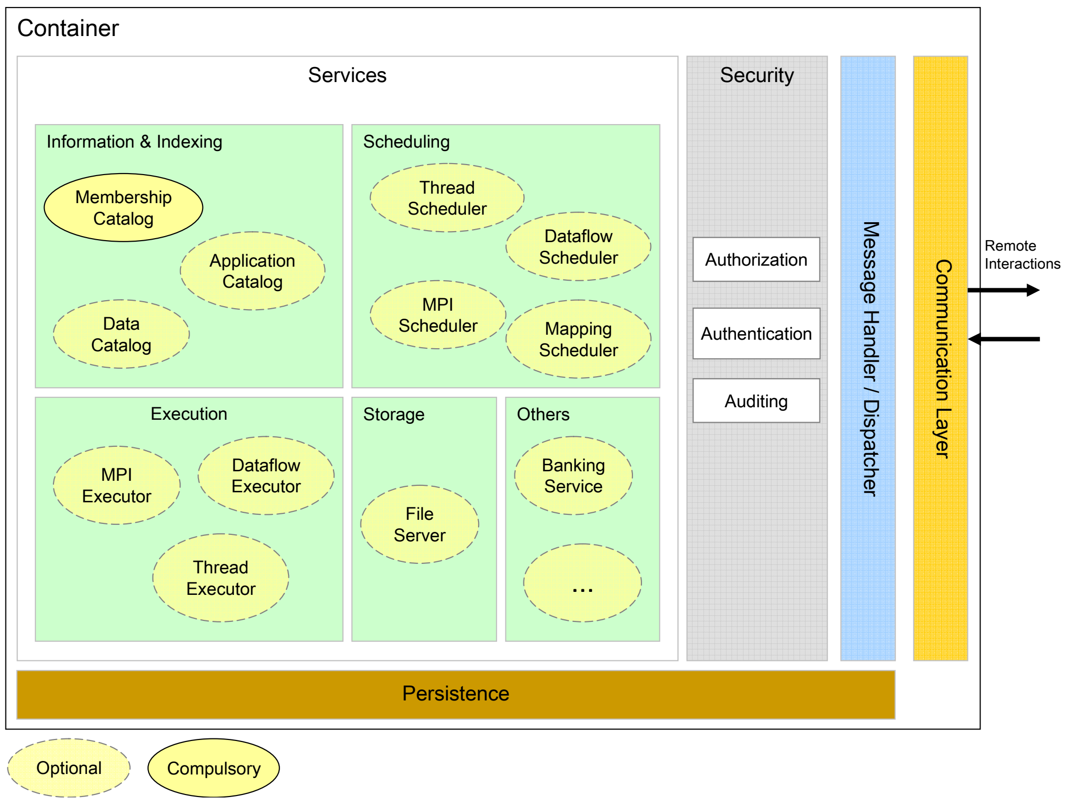


Figure 7.2: Design of Aneka container (X. Chu et al. [42]).

Figure 7.2 shows the design of the Aneka container on a single desktop node. To support scalability, the Aneka container is designed to be lightweight by providing the bare minimum functionality needed for an enterprise Grid node. It provides the base infrastructure of an enterprise Grid that consists of persistence, security (authorization, authentication and auditing), and communication (message handling and dispatching). Every service communication between nodes is treated as a message, handled and dispatched through the message handler/dispatcher that acts as a front controller. The Aneka container also hosts a compulsory Membership Catalog service which maintains network connectivity between the enterprise Grid nodes.

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 optional services are information and indexing, scheduling, execution, and storage services. This provides a flexible and extensible framework or interface for the provider to easily support various application models. Thus, resource users can seamlessly execute different types of application on an enterprise Grid.

To support reliability and flexibility, services are designed to be independent of each other in a container. A service can only interact with other services on the local node or other nodes through known interfaces. This means that a malfunctioning service will not affect other working services and/or the container. Therefore, the provider can easily configure and manage existing services or introduce new ones into a container.

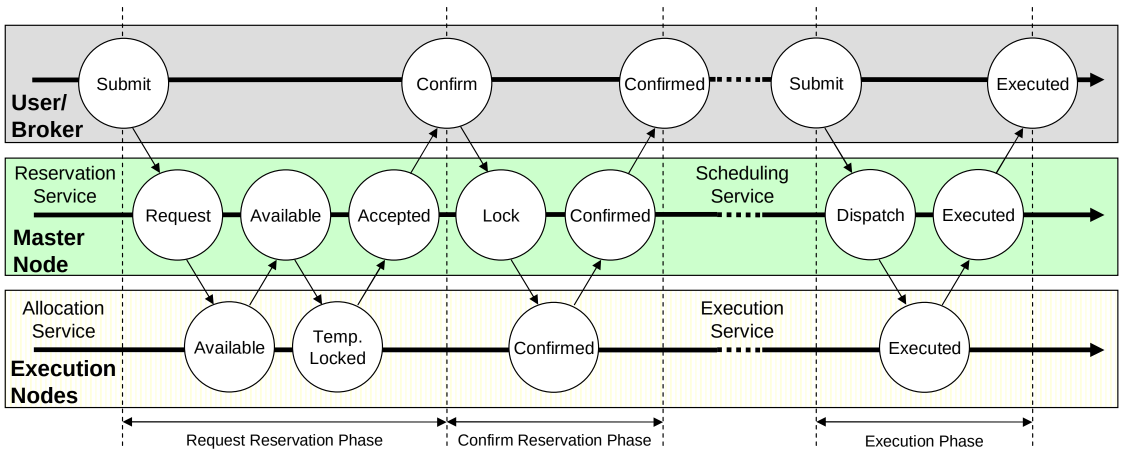


Figure 7.3: Interaction of enterprise Grid nodes.

Aneka thus provides the flexibility for the provider to implement any network architecture for an enterprise Grid. The implemented network architecture depends on the interaction of services among enterprise Grid nodes since each Aneka container on a node can directly interact with other Aneka containers reachable on the network. An enterprise Grid can have a decentralized network architecture peering individual desktop nodes directly, a hierarchical network architecture peering nodes in the hierarchy, or a centralized network architecture peering nodes through a single controller.

7.3.2 Resource Management Architecture

We implement a bi-hierarchical advance reservation mechanism for enterprise Grid with a Reservation Service at a master node that coordinates multiple execution nodes and an Allocation Service at each execution node that keeps track of the reservations at that node. This architecture was previously introduced by Venugopal et al. [118]. Figure 7.3 shows the interaction between the user/broker, the master node and execution nodes in the enterprise Grid. To use the enterprise Grid, the resource user (or a broker acting on its behalf) has to first make advanced reservations for resources required at a designated time in the future.

During the request reservation phase, the user/broker submits reservation requests through the Reservation Service at the master node. The Reservation Service discovers available execution nodes in the enterprise Grid by interacting with the Allocation Service on them. The Allocation Service at each execution node keeps track of all reservations that have been confirmed for the node and can thus check whether a new request can be satisfied or not.

By allocating reservations at each execution node instead of at the master node, computation overheads arising from making allocation decisions are distributed across multiple nodes and thus minimized, as compared to overhead accumulation at a single master node. The Reservation Service then selects the required number of execution nodes and informs their Allocation Services

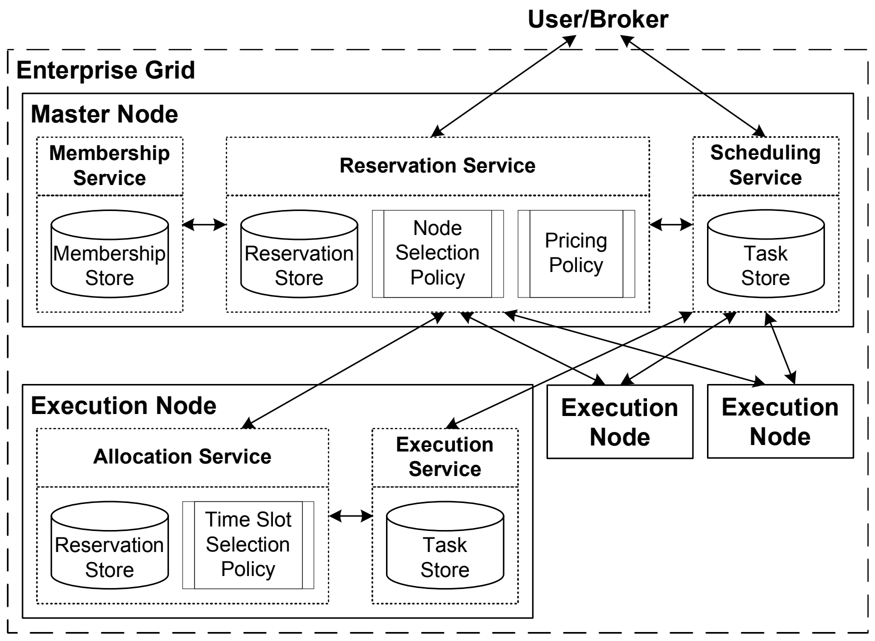


Figure 7.4: Interaction of services in enterprise Grid.

to temporarily lock the reserved time slots. After all the required reservations on the execution nodes have been temporarily locked, the Reservation Service returns the reservation outcome and its price (if successful) to the user/broker.

The user/broker may confirm or reject the reservations during the confirm reservation phase. The Reservation Service then notifies the Allocation Service of selected execution nodes to lock or remove temporarily locked time slots accordingly. We assume that a payment service is in place to ensure the user/broker has sufficient funds and can successfully deduct the required payment before the Reservation Service proceeds with the final confirmation.

During the execution phase when the reserved time arrives, the user/broker submits applications to be executed to the Scheduling Service at the master node. The Scheduling Service determines whether any of the reserved execution nodes are available before dispatching applications to them for execution, otherwise applications are queued to wait for the next available execution nodes that are part of the reservation. The Execution Service at each execution node starts executing an application after receiving it from the Scheduling Service and updates the Scheduling Service of changes in execution status. Hence, the Scheduling Service can monitor executions for an application and notify the user/broker upon completion.

7.3.3 Allocating Advanced Reservations

Figure 7.4 shows that the process of allocating advanced reservations happens in two levels: the Allocation Service at each execution node and the Reservation Service at the master node. Both services are designed to support pluggable policies so that the provider has the flexibility to easily customize and replace existing policies for different levels and/or nodes without interfering with the overall resource management architecture.

The Allocation Service determines how to schedule a new reservation at the execution node. For simplicity, we implement the same time slot selection policy for the Allocation Service at every execution node. The Allocation Service allocates the requested time slot if the slot is available. Otherwise, it assigns the next available time slot after the requested start time that can meet the required duration.

The Reservation Service performs node selection by choosing the required number of available time slots from execution nodes and administers admission control by accepting or rejecting a reservation request. It also calculates the price for a confirmed reservation based on the implemented pricing policy. Various pricing policies considered in our work are explained in Section 7.2.4. Available time slots are selected taking into account the application requirement of the user.

The application requirement considered here is the task parallelism to execute an application. A sequential application has a single task and thus needs a single processor to run, while a parallel application needs a required number of processors to concurrently run at the same time.

For a sequential application, the selected time slots need not have the same start and end times. Hence, available time slots with the lowest prices are selected first. If there are multiple available time slots with the same price, then those with the earliest start time available are selected first. This ensures that the cheapest requested time slot is allocated first if it is available. Selecting available time slots with the lowest prices first is fair and realistic. In reality, reservations that are confirmed earlier enjoy the privilege of cheaper prices, as compared to reservation requests that arrive later.

But, for a parallel application, all the selected time slots must have the same start and end times. Again, the earliest time slots (with the same start and end times) are allocated first to ensure the requested time slot is allocated first if available. If there are more available time slots (with the same start and end times) than the required number of time slots, then those with the lowest prices are selected first.

The admission control operates according to the service requirement of the user. The service requirements examined are the deadline and budget to complete an application. We assume both deadline and budget are hard constraints. Hence, a confirmed reservation must not end after the deadline and cost more than the budget. Therefore, a reservation request is not accepted if there is insufficient number of available time slots on execution nodes that ends within the deadline and the total price of the reservation costs more than the budget.

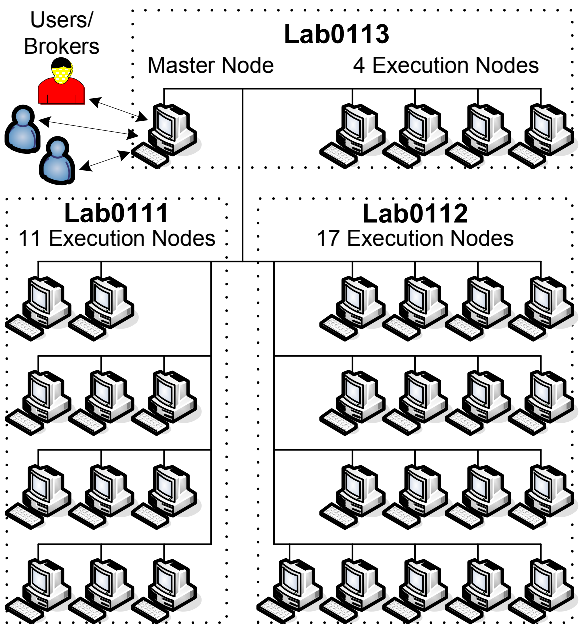


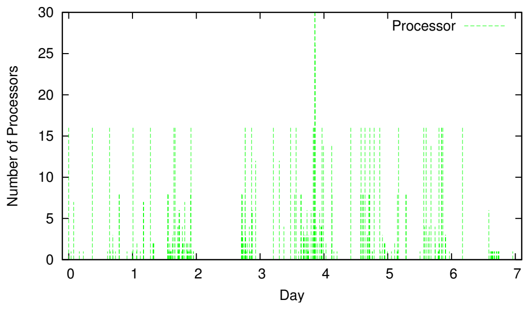
Figure 7.5: Configuration of enterprise Grid.

7.4 Performance Evaluation

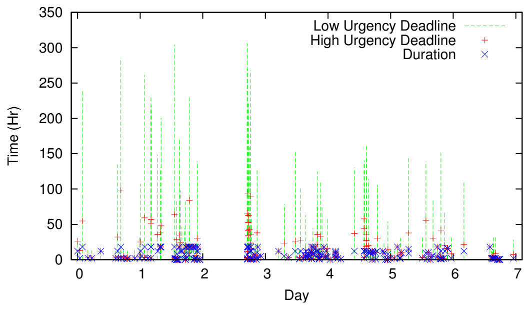
Figure 7.5 shows the enterprise Grid setup used for performance evaluation. The enterprise Grid contains 33 PCs with 1 master node and 32 execution nodes located across 3 student computer laboratories in the Department of Computer Science and Software Engineering, The University of Melbourne. Synthetic workloads are created by utilizing trace data. The experiments utilize 238 reservation requests in the last 7 days of the SDSC SP2 trace (April 1998 to April 2000) version 2.2 from Feitelson's Parallel Workload Archive [2]. The SDSC SP2 trace from the San Diego Supercomputer Center (SDSC) in USA is chosen due to the highest resource utilization of 83.2% among available traces to ideally model a heavy workload scenario.

The trace only provides the inter-arrival times of reservation requests, the number of processors to be reserved as shown in Figure 7.6(a) (downscaled from a maximum of 128 nodes in the trace to a maximum of 32 nodes), and the duration to be reserved as shown in Figure 7.6(b). However, service requirements are not available from this trace. Hence, we adopt a similar methodology in Section 3.5.1 to synthetically assign service requirements through two request classes: (i) low urgency and (ii) high urgency. Figure 7.6(b) and 7.6(c) show the synthetic values of deadline and budget for the 238 requests respectively.

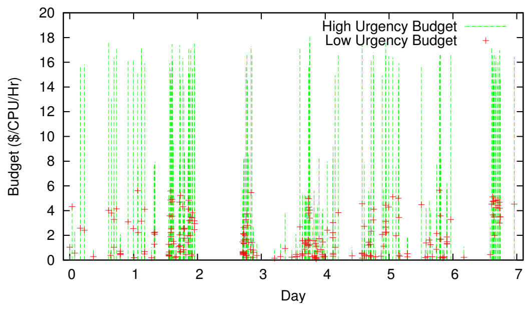
For simplicity, we only evaluate the performance of pricing for processing power as listed in Table 7.2 with various combinations of application requirements (sequential and parallel) and request classes (low urgency and high urgency). However, the performance evaluation can be easily extended to include other resource types such as memory, storage, and bandwidth as shown in Table 7.1. Both low urgency and high urgency classes are selected so as to observe the performance



(a) Number of processors (from trace)



(b) Duration (from trace) and deadline (synthetic)



(c) Budget (synthetic)

Figure 7.6: Last 7 days of SDSC SP2 trace (April 1998 to April 2000) with 238 requests.

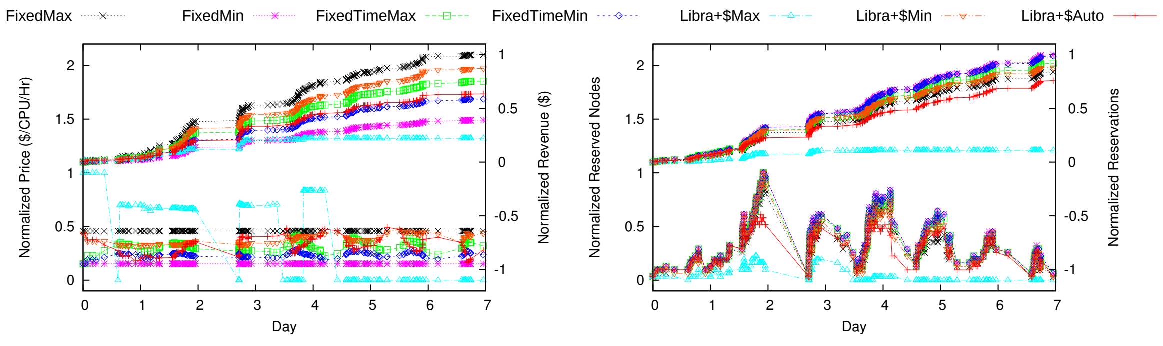
under extreme cases of service requirements with respective highest and lowest values for deadline and budget. We also assume that every user/broker can definitely accept another reservation time slot proposed by the enterprise Grid if the requested one is not possible, provided that the proposed time slot still satisfies both application and service requirements of the user.

7.5 Performance Results

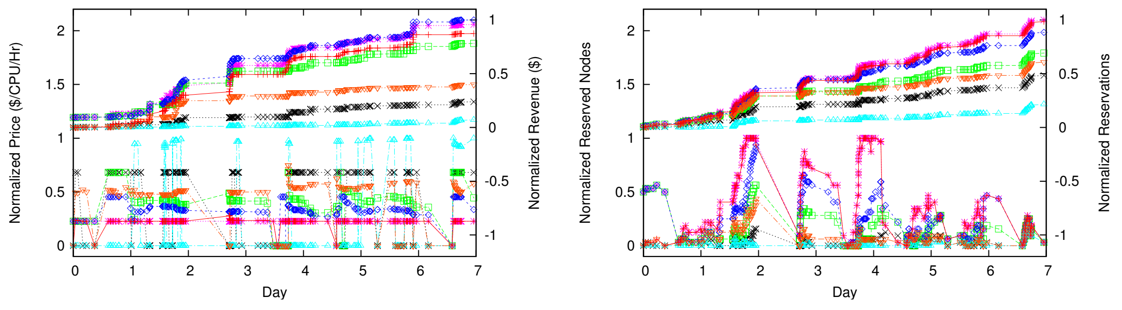
We analyze the performance results of autonomic metered pricing over 7 days with respect to the application and service requirements of users. Performance results have been normalized to produce standardized values within the range of 0 to 1 for easier comparison. The four performance metrics being measured are: (i) the price for a confirmed reservation (in $/CPU/Hr), (ii) the accumulated revenue for confirmed reservations (in $), (iii) the number of reserved nodes, and (iv) the accumulated number of confirmed reservations. The revenue of a confirmed reservation is the total sum of revenue at each of its reserved node calculated using the assigned price (depending on the specific pricing mechanism) and reserved duration at that node. Then, the price of a confirmed reservation can be computed to reflect the average price across all its reserved nodes.

7.5.1 Application Requirements

Figure 7.7 shows the performance results for low urgency requests (with long deadline and low budget). For sequential applications, all pricing mechanisms listed in Table 7.2 except Libra+$Auto

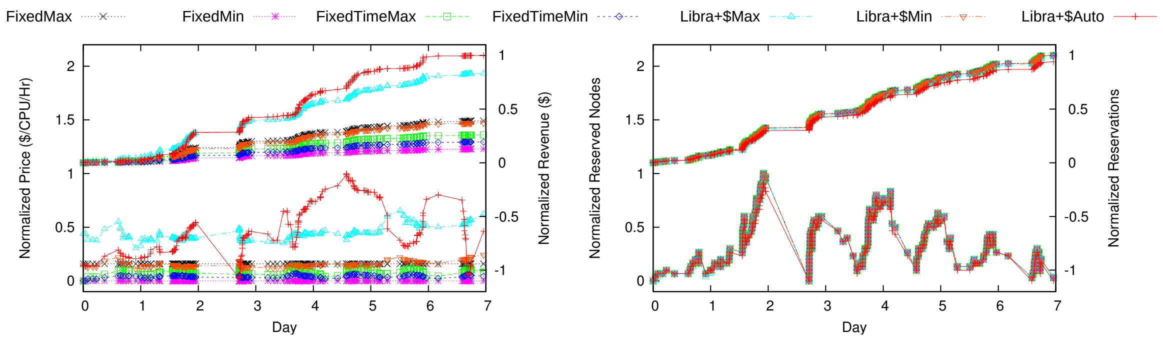


(a) Sequential: Price/Revenue (b) Sequential: Node/Reservation

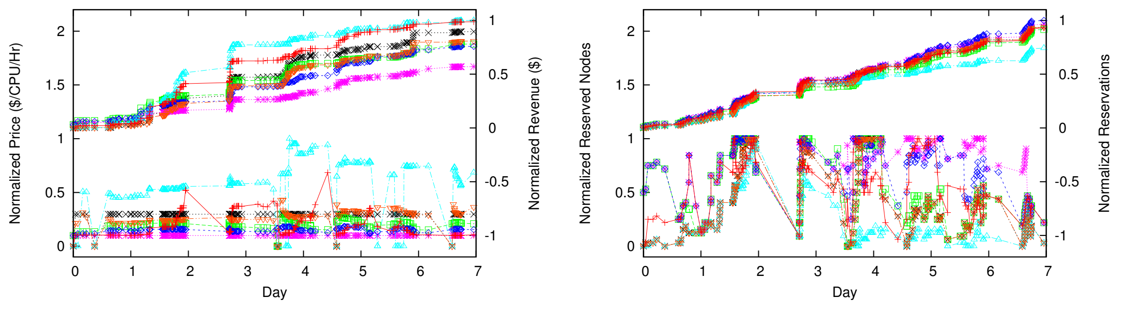


(c) Parallel: Price/Revenue (d) Parallel: Node/Reservation

Figure 7.7: Low urgency requests (long deadline and low budget).



(a) Sequential: Price/Revenue (b) Sequential: Node/Reservation



(c) Parallel: Price/Revenue (d) Parallel: Node/Reservation

Figure 7.8: High urgency requests (short deadline and high budget).

and Libra+$Max reserve and confirm relatively equal number of nodes and reservations respectively (Figure 7.7(b)). But, FixedMax provides the highest revenue, followed by Libra+$Min, FixedTimeMax, Libra+$Auto, FixedTimeMin, FixedMin, and Libra+$Max (Figure 7.7(a)).

For parallel applications, only FixedTimeMin and FixedMin provide slightly higher revenue than Libra+$Auto (Figure 7.7(c)). FixedMax, Libra+$Min, and FixedTimeMax which provide the highest revenue for sequential applications (Figure 7.7(a)), instead provide the lowest revenue for parallel applications (Figure 7.7(c)). This is because parallel applications need multiple nodes which require higher budget, compared to sequential applications which only require a single node. Hence, the low budget results in a huge inconsistency in performance between sequential and parallel applications for FixedMax, FixedMin, FixedTimeMax, FixedTimeMin, and Libra+$Min due to the inflexibility of static pricing parameters.

Libra+$Max provides the lowest revenue for both sequential and parallel applications although it achieves the highest prices at various times (Figure 7.7(a) and 7.7(c)). Again, this highlights the inflexibility of static pricing parameters to maximize revenue for different application requirements. In this case, β of Libra+$Max is set too high such that requests are rejected due to low budget. However, for parallel applications, Libra+$Auto confirms more reservations and reserves a higher number of nodes than both Libra+$Max and Libra+$Min at various times (Figure 7.7(d)) and thus provides substantially higher revenue even with more constant prices (Figure 7.7(c)). This is due to Libra+$Auto able to automatically adjust to a lower β to meet the low budget when nodes are not reserved.

Figure 7.8 shows the performance results for high urgency requests (with short deadline and high budget). For sequential applications, all pricing mechanisms listed in Table 7.2 reserve and confirm similar number of nodes and reservations respectively (Figure 7.8(b)). Libra+$Auto and Libra+$Max provide a significantly higher revenue than other mechanisms through higher prices for shorter deadlines (Figure 7.8(a)). In particular, Libra+$Auto is able to exploit the high budget by automatically adjusting to a higher β to increase prices and maximize revenue when the availability of nodes is low. Figure 7.8(a) shows that Libra+$Auto continues increasing prices to higher than that of Libra+$Max and other pricing mechanisms when demand is high such as during the later half of day 1, 2, 3, and 5. On the other hand, when demand is low such as during the early half of day 2, 3, 5, and 6, Libra+$Auto keeps reducing prices to lower than that of Libra+$Max to accept requests that are not willing to pay more.

For parallel applications, Libra+$Auto and Libra+$Max only manage marginally higher revenue than other mechanisms (Figure 7.8(c)). This is due to parallel applications needing multiple nodes which are less likely to be available for short deadlines. However, Libra+$Auto is still able to achieve similar revenue as Libra+$Max even though Libra+$Max accumulates more revenue much earlier from Day 2 to 5 (Figure 7.8(c)). This is because Libra+$Auto also adjusts to a lower β at various times to accommodate more requests with lower prices than Libra+$Max to eventually fix the initial shortfall (Figure 7.8(d)).

7.5.2 Service Requirements

Out of the four fixed pricing mechanisms listed in Table 7.2, FixedMax typically provides the highest revenue (maximum bound), followed by FixedTimeMax, FixedTimeMin, and FixedMin with the lowest revenue (minimum bound). However, there can be exceptions, such as for low urgency requests (with low budget) of parallel applications where FixedMax and FixedTimeMax provide much more lower revenue than FixedTimeMin and FixedMin due to more parallel applications needing higher budget for more nodes being rejected (Figure 7.7(c)). Nevertheless, FixedTime mechanisms is easier to derive and more reliable than Fixed mechanisms since it supports a range of prices across various time periods of resource usage. But, all four mechanisms do not consider service requirements of users such as deadline and budget.

On the other hand, Libra+$ charges a lower price for a request with longer deadline as an incentive to encourage users to submit requests with longer deadlines that are more likely to be accommodated than shorter deadlines. For a request with short deadline, Libra+$Max and Libra+$Min charge a higher price relative to their βs in Table 7.2. Libra+$Max provides higher revenue than Libra+$Min for high urgency requests (with short deadline and high budget) (Figure 7.8(a) and 7.8(c)), but lower revenue than Libra+$Min for low urgency requests (with long deadline and low budget) (Figure 7.7(a) and 7.7(c)). This is due to the statically defined high β of Libra+$Max that rejects low urgency requests with low budget.

Libra+$Auto achieves the highest revenue for high urgency requests (Figure 7.8(a) and 7.8(c)). For low urgency requests, although Libra+$Auto provides lower revenue than FixedMax, Libra+$Min, and FixedTimeMax for sequential applications (Figure 7.7(a)), it is able to gain higher revenue than them for parallel applications (Figure 7.7(c)).

7.6 Summary

We emphasize the importance of implementing autonomic metered pricing for a utility computing service to self-adjust prices to increase revenue. Through the actual implementation of an enterprise Grid, we show that a simple autonomic pricing mechanism called Libra+$Auto is able to achieve higher revenue than other fixed pricing mechanisms by considering two essential user requirements: (i) application (sequential and parallel) and (ii) service (deadline and budget). The use of advanced reservations enables Libra+$Auto to self-adjust prices in a more fine-grained manner based on the expected workload demand and availability of nodes so that more precise incentives can be offered to individual users to promote demand and thus improve revenue.

Our future work will involve conducting experimental studies using real applications and service requirements of users which can be collected by providers such as Amazon, Sun Microsystems, or

Tsunamic Technologies. We also need to understand how users will react to price changes and when they will switch providers. This knowledge can be used to derive more sophisticated models for a better autonomic pricing mechanism that considers more dynamic factors such as user response from price changes and competition from other providers. A stochastic model can then be built based on historical observation data to predict future demand and adjust prices accordingly. In addition, allowing cancellation of reservations is essential to provide more flexibility and convenience for users since user requirements can change over time. Therefore, future work needs to investigate the implication of cancellations for a utility computing service and possible overbooking of reservations to address cancellations. It may be possible to apply revenue management [90] to monitor current cancellations, amend cancellation and refund policies, and adjust prices for new reservations accordingly.

Chapter 8

Concluding Remarks

This thesis addresses the problem of enabling utility-based resource management for cluster computing to facilitate service-oriented Grid computing and utility computing. Utility-based cluster resource management considers user-centric needs such as QoS requirements so that requests are selectively accepted and prioritized accordingly in order to meet their requirements. By applying market-based mechanisms, we have shown that achieving users' needs not only returns higher value for the users, but also generates more revenue for the cluster provider. In this chapter, we summarize our contributions and present possible future research directions for this work.

8.1 Summary

In Chapter 2, we first present an abstract model and taxonomy to conceptualize the essential functions of a market-based cluster RMS and categorize various market-based RMSs that can support utility-based cluster computing in practice respectively. The taxonomy is then applied to survey existing market-based RMSs designed for both cluster and other computing platforms to not only reveal key design factors and issues that are still outstanding and crucial, but also provide insights for extending and reusing components of existing market-based RMSs. The mapping of the taxonomy to the range of market-based RMSs has successfully demonstrated that the proposed taxonomy is sufficiently comprehensive to characterize existing and future market-based RMSs for utility-based cluster computing.

In Chapter 3, a pricing function called Libra+$ is presented to satisfy essential requirements for pricing cluster resources in a commodity market model. Libra+$ adapts the price according to the changing supply and demand of resources by computing higher pricing when cluster workload increases. This not only serves as an effective means for admission control to prevent the cluster from overloading and tolerate against job runtime under-estimates, but also generates more revenue for the cluster provider. Libra+$ also provides the flexibility for the cluster provider to easily configure the pricing in order to modify the level of sharing.

In Chapter 4, we propose a job admission control called LibraSLA that examines whether accepting a new job will affect the SLA conditions of other accepted jobs, in particular, how penalties incurred on these jobs will decrease their utility. LibraSLA is able to accommodate more jobs with soft deadlines so as to enhance the utility of the cluster. This reiterates the importance of considering service needs in terms of SLAs to support utility-based cluster computing.

However, a job admission control relies heavily on accurate runtime estimates of jobs in order to prioritize jobs effectively and performs worse than expected when actual runtime estimates are inaccurate and often over estimated. Hence, in Chapter 5, we present a job admission control called LibraRiskD that can manage the risk of inaccurate runtime estimates more effectively by considering the risk of deadline delay.

With the emergence of utility computing, a computing service provider needs to focus on what objectives to achieve. Hence, in Chapter 6, we identify four essential objectives to support utility computing and apply risk analysis techniques to analyze the effectiveness of resource management policies in achieving the objectives. We demonstrate that an objective that is not achieved can severely impact on the overall achievement of other objectives. It is thus essential to examine the achievement of all key objectives together, rather than each standalone objective to correctly identify the best policy that can meet all the objectives.

Finally, in Chapter 7, we present an autonomic version of Libra+$ which is able to self-adjust prices based on the expected workload demand and availability of nodes in an enterprise Grid through advanced reservations to improve revenue. Again, we stress the need to consider application and service requirements in a utility computing service, in particular, using advanced reservations to compute prices in a more fine-grained manner so that more precise incentives can be offered to users to promote demand and thus revenue. This provides a better understanding of setting prices in metered pricing for a utility computing service, especially why the use of variable prices is preferred to fixed prices.

8.2 Future Directions

This thesis reveals seven future directions to further enhance utility-based resource management for cluster computing. As shown in Figure 8.1, these future directions can be grouped into two categories: (i) management perspective and (ii) engineering perspective. Market-based resource management, computational risk management, customer-driven service management, autonomic resource management, and service benchmarking collectively upgrades the overall management of utility-based resources, while virtual machine based systems engineering and Aneka: enterprise Grid infrastructure together improves the engineering of systems to enable utility-based resource management.

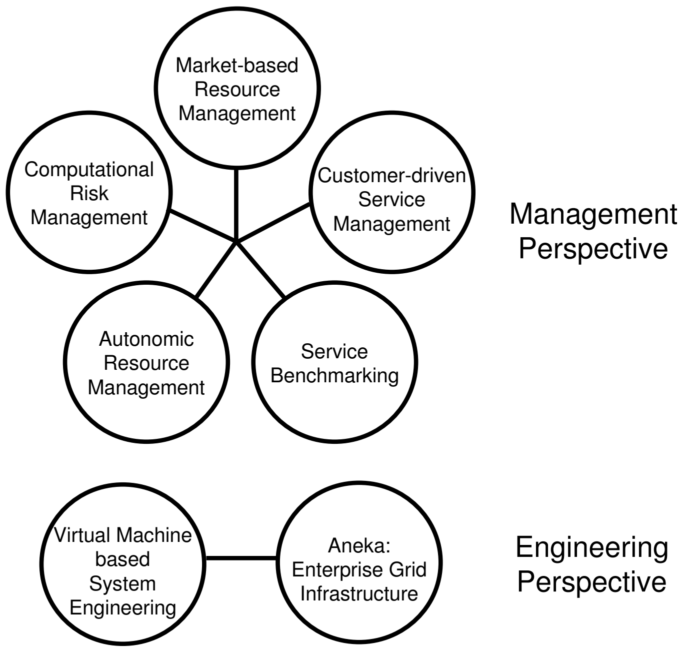

Market-based Resource Management
Computational Risk Management
Customer-driven Service Management
Management Perspective
Autonomic Resource Management
Service Benchmarking
Virtual Machine based System Engineering
Aneka: Enterprise Grid Infrastructure
Engineering Perspective

Figure 8.1: Future directions.

8.2.1 Market-based Resource Management

This thesis has examined two possible market-based models for resource management: (i) commodity market model and (ii) bid-based model. But, there are many other market-based models [28] that may be applicable and thus need to be explored. For instance, adopting models that enable users to negotiate prices, such as bargaining and tendering/contract-net may encourage more users to patronize a service provider. There is also the possibility of integrating these market models with computational risk management and customer-driven service management (as mentioned below) to enhance utility-based cluster resource management.

8.2.2 Computational Risk Management

In Chapter 6, we have presented examples of how various elements of utility-based cluster resource management can be perceived as risks and hence identified risk analysis [47][83] from the field of economics as a probable solution to evaluate these risks. However, the entire risk management process comprises of many steps and thus needs to be studied thoroughly so as to fully apply its effectiveness in managing risks.

8.2.3 Customer-driven Service Management

In Chapter 6, we have highlighted customer satisfaction as a crucial success factor to excel in the service industry and thus proposed three user-centric objectives in the context of a computing service provider that can lead to customer satisfaction. But, there are many service quality factors that can influence customer satisfaction [101][116]. Factors which provide personalized attention to customers include enabling communication to keep customers informed and obtain feedback from them, increasing access and approachability to customers, and understanding specific needs of customers. Other factors that encourage trust and confidence in customers are security measures undertaken against risks and doubts, credibility of provider, and courtesy towards customers. Therefore, a detailed study of all possible customer characteristics needs to be done to determine whether a cluster RMS needs to consider more relevant characteristics to better support utility-based resource management.

8.2.4 Autonomic Resource Management

In Chapter 7, we have showcased the benefits of autonomic pricing which self-adjusts prices for utility-based resource management. But, we currently assume that reservations cannot be cancelled after they have been confirmed. This stringent limitation has to be overcome since user requirements can change over time and thus may require cancellation of reservations. In other words, cluster RMSs must be able to self-manage the reservation progress continuously by monitoring current cancellations, amending cancellation and refund policies, and adjusting prices for new reservations accordingly. There is even the possibility of overbooking resources to address cancellations. There are also other aspects of autonomy, such as self-adapting schedules to meet changes in existing service requirements and self-configuring components to satisfy new service requirements. Hence, more autonomic and intelligent cluster RMSs are essential to effectively manage the limited supply of resources with dynamically changing service demand. For users, there can be brokering systems acting on their behalf to select the most suitable providers and negotiate with them to achieve the most ideal service contracts. Thus, providers also require autonomic resource management to selectively choose the appropriate requests to accept and execute depending on a number of operating factors, such as the expected availability and demand of services (both current and future), and existing service obligations.

8.2.5 Service Benchmarking

For the performance evaluation of various resource management policies, this thesis has employed traces from Feitelson's Parallel Workload Archive [2] and probability distributions to model application and service requirements respectively. This is because there are currently no service benchmarks available to evaluate utility-based resource management for cluster computing in a standard manner. Moreover, there can be different emphasis of application requirements such as data-intensive and workflow applications, and service requirements such as reliability and trust/security. Therefore, it is necessary to derive a standard set of service benchmarks for the accurate evaluation of resource management policies. The benchmarks should be able to reflect realistic application and service requirements of users that can in turn faciliate the forcasting and prediction of future users' needs.

8.2.6 Virtual Machine based System Engineering

Recently, virtualization [19] has enabled the abstraction of computing resources such that a single physical machine is able to function as multiple logical Virtual Machines (VMs). A key benefit of VMs is the ability to host multiple operating system environments which are completely isolated from one another on the same physical machine. Another benefit is the capability to configure VMs to utilize different partitions of resources on the same physical machine. For example, on a physical machine, one VM can be allocated 10% of the processing power, while another VM can be allocated 20% of the processing power. Hence, VMs can be started and stopped dynamically to meet the changing demand of resources by users as opposed to limited resources on a physical machine. In particular, VMs may be assigned various resource management policies catering to different user needs and demands to better support the implementation of utility-based resource allocation.

8.2.7 Aneka: Enterprise Grid Infrastructure

In Chapter 7, we use a service-oriented enterprise Grid platform called Aneka [42] to implement an autonomic pricing mechanism using advanced reservations. But, Aneka can be further extended to improve utility-based resource allocation. One possible extension is to support the creation of multiple Aneka nodes using VMs on a single physical machine, instead of the creation of a single Aneka node on a physical machine. This in turn requires proper interfacing and coordination to differentiate between Aneka nodes on VMs and physical machines so that these Aneka nodes can still function seamlessly without the user being aware of the underlying differences. New services can also be developed to exploit the virtualization of Aneka nodes, such as clustering nodes for specific usages, changing policies automatically based on operating conditions, and exploiting specific virtual resources for different application types.

References

[1] Business Process Execution Language for Web Services version 1.1, July 2005. http://www.ibm.com/developerworks/library/specification/ws-bpel/.

[2] Parallel Workloads Archive, May 2005. http://www.cs.huji.ac.il/labs/parallel/workload/.

[3] Top500 Supercomputing Sites, May 2005. http://www.top500.org.

[4] Web Services Agreement Specification (WS-Agreement) version 1.1 draft 18, July 2005. http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap-agreement.pdf.

[5] J. Abate, P. Wang, and K. Sepehrnoori. Parallel Compositional Reservoir Simulation on Clusters of PCs. *International Journal of High Performance Computing Applications*, 15(1):13–21, Feb. 2001.

[6] D. Abramson, R. Buyya, and J. Giddy. A Computational Economy for Grid Computing and its Implementation in the Nimrod-G Resource Broker. *Future Generation Computer Systems*, 18(8):1061–1074, Oct. 2002.

[7] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A Tool for Performing Parametrised Simulations using Distributed Workstations. In *Proceedings of the 4th International Symposium on High Performance Distributed Computing (HPDC4)*, pages 112–121, Pentagon City, VA, USA, Aug. 1995. IEEE Computer Society: Los Alamitos, CA, USA.

[8] V. Akcelik, J. Bielak, G. Biros, I. Epanomeritakis, A. Fernandez, O. Ghattas, E. J. Kim, J. Lopez, D. O'Hallaron, T. Tu, and J. Urbanic. High Resolution Forward And Inverse Earthquake Modeling on Terascale Computers. In *Proceedings of the 16th Supercomputing Conference (SC 2003)*, Phoenix, AZ, USA, Nov. 2003. IEEE Computer Society: Los Alamitos, CA, USA.

[9] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. SuperWeb: Research Issues in Java-based Global Computing. *Concurrency: Practice and Experience*, 9(6):535–553, June 1997.

[10] Altair Grid Technologies. *OpenPBS Release 2.3 Administrator Guide*, Aug. 2000. http://www.openpbs.org/docs.html.

[11] Amazon. Elastic Compute Cloud (EC2), Jan. 2007. http://www.amazon.com/ec2/.

[12] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren. An Opportunity Cost Approach for Job Assignment in a Scalable Computing Cluster. *IEEE Transactions on Parallel and Distributed Systems*, 11(7):760–768, July 2000.

[13] Y. Amir, B. Awerbuch, and R. S. Borgstrom. A Cost-Benefit Framework for Online Management of a Metacomputing System. In *Proceedings of the 1st International Conference on Information and Computation Economies (ICE '98)*, pages 140–147, Charleston, SC, USA, Oct. 1998. ACM Press: New York, NY, USA.

[14] A. Anastasiadi, S. Kapidakis, C. Nikolaou, and J. Sairamesh. A Computational Economy for Dynamic Load Balancing and Data Replication. In *Proceedings of the 1st International Conference on Information and Computation Economies (ICE '98)*, pages 166–180, Charleston, SC, USA, Oct. 1998. ACM Press: New York, NY, USA.

[15] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, Feb. 1995.

[16] A. AuYoung, B. N. Chun, A. C. 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 2004)*, Boston, MA, USA, Oct. 2004.

[17] M. Backschat, A. Pfaffinger, and C. Zenger. Economic-Based Dynamic Load Distribution in Large Workstation Networks. In *Proceedings of the 2nd International Euro-Par Conference (Euro-Par 1996)*, volume 1124/1996 of *Lecture Notes in Computer Science (LNCS)*, pages 631–634, Lyon, France, Aug. 1996. Springer-Verlag: Heidelberg, Germany.

[18] A. Barak and O. La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Future Generation Computer Systems*, 13(4–5):361–372, Mar. 1998.

[19] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, Bolton Landing, NY, USA, Oct. 2003. ACM Press: New York, NY, USA.

[20] A. Barmouta and R. Buyya. GridBank: A Grid Accounting Services Architecture (GASA) for Distributed Systems Sharing and Integration. In *Proceedings of the 3rd Workshop on Internet Computing and E-Commerce (ICEC 2003), 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, Nice, France, Apr. 2003. IEEE Computer Society: Los Alamitos, CA, USA.

[21] L. A. Barroso, J. Dean, and U. Hlzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, Mar.–Apr. 2003.

[22] M. Bhandarkar, L. V. Kalé, E. de Sturler, and J. Hoeflinger. Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science (ICCS 2001)*, volume 2074/2001 of *Lecture Notes in Computer Science (LNCS)*, pages 108–117, San Francisco, CA, USA, May 2001. Springer-Verlag: Heidelberg, Germany.

[23] T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao. A Taxonomy for Describing Matching and Scheduling Heuristics for Mixed-Machine Heterogeneous Computing systems. In *Proceedings of the 17th Symposium on Reliable Distributed Systems*, pages 330–335, West Lafayette, IN, USA, Oct. 1998. IEEE Computer Society: Los Alamitos, CA, USA.

[24] J. Bredin, D. Kotz, and D. Rus. Utility Driven Mobile-Agent Scheduling. Technical Report PCS-TR98-331, Department of Computer Science, Dartmouth College, Oct. 1998.

[25] R. Buyya, editor. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[26] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. In *Proceedings of the 4th International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2000)*, volume 1, pages 283–289, Beijing, China, May 2000. IEEE Computer Society: Los Alamitos, CA, USA.

[27] R. Buyya, D. Abramson, and J. Giddy. A Case for Economy Grid Architecture for Service Oriented Grid Computing. In *Proceedings of the 10th International Heterogeneous Computing Workshop (HCW 2001)*, San Francisco, CA, USA, Apr. 2001. IEEE Computer Society: Los Alamitos, CA, USA.

[28] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic Models for Resource Management and Scheduling in Grid Computing. *Concurrency and Computation: Practice and Experience*, 14(13–15):1507–1542, Nov.–Dec. 2002.

[29] R. Buyya, D. Abramson, and S. Venugopal. The Grid Economy. *Proceedings of the IEEE*, 93(3):698–714, Mar. 2005.

[30] R. Buyya, T. Cortes, and H. Jin. Single System Image. *International Journal of High Performance Computing Applications*, 15(2):124–135, May 2001.

[31] R. Buyya, J. Giddy, and D. Abramson. An Evaluation of Economy-based Resource Trading and Scheduling on Computational Power Grids for Parameter Sweep Applications. In *Proceedings of the 2nd Annual Workshop on Active Middleware Services (AMS 2000)*, Pittsburgh, PA, USA, Aug. 2000. Kluwer Academic Publishers: Dordrecht, Netherlands.

[32] 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*, 14(13–15):1175–1220, Nov.–Dec. 2002.

[33] R. Buyya, M. Murshed, and D. Abramson. A Deadline and Budget Constrained Cost-Time Optimization Algorithm for Scheduling Task Farming Applications on Global Grids. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2002)*, Las Vegas, NV, USA, June 2002. CSREA Press: USA.

[34] A. Byde, M. Sallé, and C. Bartolini. Market-Based Resource Allocation for Utility Data Centers. Technical Report HPL-2003-188, HP Labs, Bristol, Sept. 2003.

[35] T. L. Casavant and J. G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, Feb. 1988.

[36] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In *Proceedings of the 12th International Symposium on High Performance Distributed Computing (HPDC12)*, pages 90–100, Seattle, WA, USA, June 2003. IEEE Computer Society: Los Alamitos, CA, USA.

[37] C. Chen, M. Maheswaran, and M. Toulouse. Supporting Co-allocation in an Auctioning-based Resource Allocator for Grid Systems. In *Proceedings of the 11th International Heterogeneous Computing Workshop (HCW 2002)*, Fort Lauderdale, FL, USA, Apr. 2002. IEEE Computer Society: Los Alamitos, CA, USA.

[38] M. Chen, G. Yang, and X. Liu. Gridmarket: A Practical, Efficient Market Balancing Resource for Grid and P2P Computing. In *Proceedings of the 2nd International Workshop on Grid and Cooperative Computing (GCC 2003)*, volume 3033/2004 of *Lecture Notes in Computer Science (LNCS)*, pages 612–619, Shanghai, China, Dec. 2003. Springer-Verlag: Heidelberg, Germany.

[39] Y. Chen, A. Das, N. Gautama, Q. Wang, and A. Sivasubramaniam. Pricing-based strategies for autonomic control of web servers for time-varying request arrivals. *Engineering Applications of Artificial Intelligence*, 17(7):841–854, Oct. 2004.

[40] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and Performance of an Enterprise Desktop Grid System. *Journal of Parallel and Distributed Computing*, 63(5):597–610, May 2003.

[41] A. Chien, S. Parkin, M. Lauria, M. Buchanan, K. Hane, L. Giannini, and J. Prusakova. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing (PP 1997)*, Minneapolis, MN, USA, Mar. 1997. SIAM: Philadelphia, PA, USA.

[42] X. Chu, K. Nadiminti, C. Jin, S. Venugopal, and R. Buyya. Aneka: Next-Generation Enterprise Grid Platform for e-Science and e-Business Applications. In *Proceedings of the 3th International Conference on e-Science and Grid Computing (e-Science 2007)*, pages 151–159, Bangalore, India, Dec. 2007. IEEE Computer Society: Los Alamitos, CA, USA.

[43] B. N. Chun and D. E. Culler. Market-based Proportional Resource Sharing for Clusters. Technical Report CSD-1092, Computer Science Division, University of California at Berkeley, Jan. 2000.

[44] B. N. Chun and D. E. Culler. REXEC: A Decentralized, Secure Remote Execution Environment for Clusters. In *Proceedings of the 4th Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC 2000)*, volume 1797/2000 of *Lecture Notes in Computer Science (LNCS)*, Toulouse, France, Jan. 2000. Springer-Verlag: Heidelberg, Germany.

[45] B. N. Chun and D. E. Culler. User-centric Performance Analysis of Market-based Cluster Batch Schedulers. In *Proceedings of the 2nd International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, pages 22–30, Berlin, Germany, May 2002. IEEE Computer Society: Los Alamitos, CA, USA.

[46] B. F. Cooper and H. Garcia-Molina. Bidding for Storage Space in a Peer-to-Peer Data Preservation System. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, pages 372–381, Vienna, Austria, July 2002. IEEE Computer Society: Los Alamitos, CA, USA.

[47] M. Crouhy, D. Galai, and R. Mark. *The Essentials of Risk Management.* McGraw-Hill, New York, NY, USA, 2006.

[48] J. P. Degabriele and D. Pym. Economic Aspects of a Utility Computing Service. Technical Report HPL-2007-101, HP Labs, Bristol, July 2007.

[49] I. Ekmecić, I. Tartalja, and V. Milutinović. EM^3: A Taxonomy of Heterogeneous Computing Systems. *IEEE Computer*, 28(12):68–70, Dec. 1995.

[50] I. Ekmecić, I. Tartalja, and V. Milutinović. A Survey of Heterogeneous Computing: Concepts and Systems. *Proceedings of the IEEE*, 84(8):1127–1144, Aug. 1996.

[51] Y. Etsion and D. Tsafrir. A Short Survey of Commercial Cluster Batch Schedulers. Technical Report 2005-13, School of Computer Science and Engineering, The Hebrew University of Jerusalem, May 2005.

[52] T. Eymann, M. Reinicke, O. Ardaiz, P. Artigas, F. Freitag, and L. Navarro. Decentralized Resource Allocation in Application Layer Networks. In *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, pages 645–650, Tokyo, Japan, May 2003. IEEE Computer Society: Los Alamitos, CA, USA.

[53] D. F. Ferguson, Y. Yemini, and C. Nikolaou. Microeconomic Algorithms for Load Balancing in Distributed Computer Systems. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS '88)*, pages 491–499, San Jose, CA, USA, June 1988. IEEE Computer Society: Los Alamitos, CA, USA.

[54] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, San Francisco, CA, USA, 2003.

[55] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *Proceedings of the 7th International Workshop on Quality of Service (IWQoS 1999)*, pages 27–36, London, UK, June 1999. IEEE Communications Society: Piscataway, NJ, USA.

[56] W. Gentzsch. Grid Computing in Industry. *CSI Communications*, 29(1):30–34, July 2005.

[57] C. Gribble, X. Cavin, M. Hartner, and C. Hansen. Cluster-based Interactive Volume Rendering with Simian. Technical Report UUCS-03-017, School of Computing, University of Utah, Sept. 2003.

[58] W. Gropp, E. Lusk, and T. Sterling, editors. *Beowulf Cluster Computing with Linux.* MIT Press, Cambridge, MA, USA, second edition, 2003.

[59] D. Grosu and A. Das. Auction-Based Resource Allocation Protocols in Grids. In *Proceedings of the 16th International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*, pages 20–27, Cambridge, MA, USA, Nov. 2004. ACTA Press: Calgary, Canada.

[60] HP. Utility Computing, Oct. 2007. http://www.hp.com/go/utility/.

[61] K.-W. Huang and A. Sundararajan. Pricing Models for On-Demand Computing. Working Paper CeDER-05-26, New York University, Nov. 2005.

[62] IBM. On Demand Business, Nov. 2004. http://www.ibm.com/ondemand/.

[63] D. E. Irwin, L. E. Grit, and J. S. Chase. Balancing Risk and Reward in a Market-based Task Service. In *Proceedings of the 13th International Symposium on High Performance Distributed Computing (HPDC13)*, pages 160–169, Honolulu, HI, USA, June 2004. IEEE Computer Society: Los Alamitos, CA, USA.

[64] M. Islam, P. Balaji, P. Sadayappan, and D. K. Panda. Towards Provision of Quality of Service Guarantees in Job Scheduling. In *Proceedings of the 6th International Conference on Cluster Computing (Cluster 2004)*, pages 245–254, San Diego, CA, USA, Sept. 2004. IEEE: Piscataway, NJ, USA.

[65] S. M. Jackson. *Allocation Management with QBank*. Pacific Northwest National Laboratory, 2004. http://www.emsl.pnl.gov/docs/mscf/qbank/.

[66] H. Jin. ChinaGrid: Making Grid Computing a Reality. In *Proceedings of the 7th International Conference on Asian Digital Libraries (ICADL 2004)*, volume 3334/2004 of *Lecture Notes in Computer Science (LNCS)*, pages 13–24, Shanghai, China, Dec. 2004. Springer-Verlag: Heidelberg, Germany.

[67] L. V. Kalé and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming Using C++*, pages 175–213. MIT Press, Cambridge, MA, USA, 1996.

[68] L. V. Kalé, S. Kumar, and J. DeSouza. A Malleable-Job System for Timeshared Parallel Machines. In *Proceedings of the 2nd International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, pages 215–222, Berlin, Germany, May 2002. IEEE Computer Society: Piscataway, NJ, USA.

[69] L. V. Kalé, S. Kumar, M. Potnuru, J. DeSouza, and S. Bandhakavi. Faucets: Efficient Resource Allocation on the Computational Grid. In *Proceedings of the 33rd International Conference on Parallel Processing (ICPP 2004)*, volume 1, pages 396–405, Montreal, Canada, Aug. 2004. IEEE Computer Society: Los Alamitos, CA, USA.

[70] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. F. Skovira. *Workload Management with LoadLeveler.* IBM Redbooks, Poughkeepsie, NY, USA, 2001. http://www.redbooks.ibm.com/redbooks/pdfs/sg246038.pdf.

[71] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11(1):57–81, Mar. 2003.

[72] S. D. Kleban and S. H. Clearwater. Computation-at-Risk: Assessing Job Portfolio Management Risk on Clusters. In *Proceedings of the 3rd International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO 2004), 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, USA, Apr. 2004. IEEE Computer Society: Los Alamitos, CA, USA.

[73] S. D. Kleban and S. H. Clearwater. Computation-at-Risk: Employing the Grid for Computational Risk Management. In *Proceedings of the 6th International Conference on Cluster Computing (Cluster 2004)*, pages 347–352, San Diego, CA, USA, Sept. 2004. IEEE: Piscataway, NJ, USA.

[74] K. Krauter, R. Buyya, and M. Maheswaran. A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing. *Software: Practice and Experience*, 32(2):135–164, Feb. 2002.

[75] J. F. Kurose and R. Simha. A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems. *IEEE Transactions on Computers*, 38(5):705–717, May 1989.

[76] K. Lai, B. A. Huberman, and L. Fine. Tycoon: A Distributed Market-based Resource Allocation System. Technical Report cs.DC/0404013, HP Labs, Palo Alto, Apr. 2004.

[77] S. Lalis and A. Karipidis. JaWS: An Open Market-Based Framework for Distributed Computing over the Internet. In *Proceedings of the 1st International Workshop on Grid Computing (Grid 2000)*, volume 1971/2000 of *Lecture Notes in Computer Science (LNCS)*, pages 36–46, Bangalore, India, Dec. 2000. Springer-Verlag: Heidelberg, Germany.

[78] M. Lamanna. The LHC Computing Grid Project at CERN. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 534(1–2):1–6, Nov. 2004.

[79] D. A. Lifka. The ANL/IBM SP Scheduling system. In *Proceedings of the 1st Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 1995)*, volume 949/1995 of *Lecture Notes in Computer Science (LNCS)*, pages 295–303, Santa Barbara, CA, USA, Apr. 1995. Springer-Verlag: Heidelberg, Germany.

[80] T. W. Malone, R. E. Fikes, K. R. Grant, and M. T. Howard. Enterprise: A Market-like Task Scheduler for Distributed Computing Environments. In B. A. Huberman, editor, *The Ecology of Computation*, pages 177–205. Elsevier Science Publisher, New York, NY, USA, 1988.

[81] S. Matsuoka, S. Shimojo, M. Aoyagi, S. Sekiguchi, H. Usami, and K. Miura. Japanese Computational Grid Research Project: NAREGI. *Proceedings of the IEEE*, 93(3):522–533, Mar. 2005.

[82] M. S. Miller and K. E. Drexler. Incentive Engineering for Computational Resource Management. In B. A. Huberman, editor, *The Ecology of Computation*, chapter 10, pages 231–266. Elsevier Science Publisher, New York, NY, USA, 1988.

[83] R. R. Moeller. *COSO Enterprise Risk Management: Understanding the New Integrated ERM Framework*. John Wiley and Sons, Hoboken, NJ, USA, 2007.

[84] A. W. Mu'alem and D. G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, June 2001.

[85] T. T. Nagle and J. E. Hogan. *The Strategy and Tactics of Pricing: A Guide to Growing More Profitably*. Prentice Hall, Upper Saddle River, NJ, USA, fourth edition, 2006.

[86] P. Padala, C. Harrison, N. Pelfort, E. Jansen, M. P. Frank, and C. Chokkareddy. OCEAN: The Open Computation Exchange and Arbitration Network, A Market Approach to Meta Computing. In *Proceedings of the 2nd International Symposium on Parallel and Distributed Computing (ISPDC 2003)*, pages 185–192, Ljubljana, Slovenia, Oct. 2003. IEEE Computer Society: Los Alamitos, CA, USA.

[87] G. A. Paleologo. Price-at-Risk: A Methodology for Pricing Utility Computing Services. *IBM Systems Journal*, 43(1):20–31, 2004.

[88] B. P. Pashigian. *Price Theory and Applications*. Irwin/McGraw-Hill, Boston, MA, USA, second edition, 1998.

[89] G. F. Pfister. *In Search of Clusters*. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, 1998.

[90] R. L. Phillips. *Pricing and Revenue Optimization.* Stanford University Press, Stanford, CA, USA, 2005.

[91] Platform Computing. *LSF Version 4.1 Administrator's Guide*, 2001. http://www.platform.com/services/support/.

[92] F. I. Popovici and J. Wilkes. Profitable Services in an Uncertain World. In *Proceedings of the 18th Supercomputing Conference (SC 2005)*, Seattle, WA, USA, Nov. 2005. IEEE Computer Society: Los Alamitos, CA, USA.

[93] C. Preist, A. Byde, and C. Bartolini. Economic Dynamics of Agents in Multiple Auctions. In *Proceedings of the 5th International Conference on Autonomous Agents (AGENTS 2001)*, pages 545–551, Montreal, Canada, May–June 2001. ACM Press: New York, NY, USA.

[94] O. Rana, M. Warnier, T. B. Quillinan, F. Brazier, and D. Cojocarasu. Managing Violations in Service Level Agreements. In *Proceedings of the Usage of Service Level Agreements in Grids Workshop, 8th International Conference on Grid Computing (Grid 2007)*, Austin, TX, USA, Sept. 2007.

[95] R. Ranjan, R. Buyya, and A. Harwood. A Case for Cooperative and Incentive-Based Coupling of Distributed Clusters. In *Proceedings of the 7th International Conference on Cluster Computing (Cluster 2005)*, Boston, MA, USA, Sept. 2005. IEEE Computer Society: Los Alamitos, CA, USA.

[96] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: Accountable Execution of Untrusted Programs. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 136–141, Rio Rico, AZ, USA, Mar. 1999. IEEE Computer Society: Los Alamitos, CA, USA.

[97] D. A. Reed. Grids, the TeraGrid, and Beyond. *Computer*, 36(1):62–68, Jan. 2003.

[98] O. Regev and N. Nisan. The POPCORN Market - an Online Market for Computational Resources. In *Proceedings of the 1st International Conference on Information and Computation Economies (ICE '98)*, pages 148–157, Charleston, SC, USA, Oct. 1998. ACM Press: New York, NY, USA.

[99] H. G. Rotithor. Taxonomy of Dynamic Task Scheduling Schemes in Distributed Computing Systems. *IEE Proceedings of Computers and Digital Techniques*, 141(1):1–10, Jan. 1994.

[100] G. Sabin, G. Kochhar, and P. Sadayappan. Job Fairness in Non-Preemptive Job Scheduling. In *Proceedings of the 33rd International Conference on Parallel Processing (ICPP 2004)*, volume 1, pages 186–194, Montreal, Canada, Aug. 2004. IEEE Computer Society: Los Alamitos, CA, USA.

[101] B. Schneider and S. S. White. *Service Quality: Research Perspectives.* Sage Publications, Thousand Oaks, CA, USA, 2004.

[102] J. Sherwani, N. Ali, N. Lotia, Z. Hayat, and R. Buyya. Libra: A Computational Economy-based Job Scheduling System for Clusters. *Software: Practice and Experience*, 34(6):573–590, May 2004.

[103] J. Shneidman, C. Ng, D. C. Parkes, A. AuYoung, A. C. Snoeren, A. Vahdat, and B. N. Chun. Why Markets Could (But Don't Currently) Solve Resource Allocation Problems in Systems. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa FE, NM, USA, June 2005.

[104] T. L. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A Parallel Workstation For Scientific Computation. In *Proceedings of the 24th International Conference on Parallel Processing (ICPP 1995)*, volume 1, pages 11–14, Oconomowoc, WI, USA, Aug. 1995. CRC Press: Boca Raton, FL, USA.

[105] I. Stoica, H. Abdel-Wahab, and A. Pothen. A Microeconomic Scheduler for Parallel Computers. In *Proceedings of the 1st Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 1995)*, volume 949/1995 of *Lecture Notes in Computer Science (LNCS)*, pages 200–218, Santa Barbara, CA, USA, Apr. 1995. Springer-Verlag: Heidelberg, Germany.

[106] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. An Economic Paradigm for Query Processing and Data Migration in Mariposa. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS '94)*, pages 58–67, Austin, TX, USA, Sept. 1994. IEEE Computer Society: Los Alamitos, CA, USA.

[107] A. Sulistio, K. H. Kim, and R. Buyya. Using Revenue Management to Determine Pricing of Reservations. In *Proceedings of the 3th International Conference on e-Science and Grid Computing (e-Science 2007)*, pages 396–404, Bangalore, India, Dec. 2007. IEEE Computer Society: Los Alamitos, CA, USA.

[108] Sun Microsystems. *Sun ONE Grid Engine, Administration and User's Guide*, Oct. 2002. http://gridengine.sunsource.net/project/gridengine/documentation.html.

[109] Sun Microsystems. Sun Grid, Nov. 2004. http://www.sun.com/service/sungrid/.

[110] Supercluster Research and Development Group. *Maui Scheduler Version 3.2 Administrator's Guide*, 2004. http://www.supercluster.org/mauidocs/mauiadmin.shtml.

[111] M. Taiji, T. Narumi, Y. Ohno, N. Futatsugi, A. Suenaga, N. Takada, and A. Konagaya. Protein Explorer: A Petaflops Special-Purpose Computer System for Molecular Dynamics Simulations. In *Proceedings of the 16th Supercomputing Conference (SC 2003)*, Phoenix, AZ, USA, Nov. 2003. IEEE Computer Society: Los Alamitos, CA, USA.

[112] Z. Tari, J. Broberg, A. Y. Zomaya, and R. Baldoni. A Least Flow-Time First Load Sharing Approach for Distributed Server Farm. *Journal of Parallel and Distributed Computing*, 65(7):832–842, July 2005.

[113] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Modeling User Runtime Estimates. In *Proceedings of the 11th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2005)*, volume 3834/2005 of *Lecture Notes in Computer Science (LNCS)*, pages 1–35, Cambridge, MA, USA, June 2005. Springer-Verlag: Heidelberg, Germany.

[114] Tsunamic Technologies. Cluster On Demand, Oct. 2007. http://www.tsunamictechnologies.com/services.htm#COD.

[115] University of Wisconsin-Madison. *Condor Version 6.7.1 Manual*, 2004. http://www.cs.wisc.edu/condor/manual/v6.7/.

[116] B. Van Looy, P. Gemmel, and R. Van Dierdonck, editors. *Services Management: An Integrated Approach.* Financial Times Prentice Hall, Harlow, England, second edition, 2003.

[117] 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*, 18(6):685–699, May 2006.

[118] S. Venugopal, X. Chu, and R. Buyya. A Negotiation Mechanism for Advance Resource Reservations using the Alternate Offers Protocol. In *Proceedings of the 16th International Workshop on Quality of Service (IWQoS 2008)*, pages 40–49, Enschede, The Netherlands, June 2008. IEEE: Piscataway, NJ, USA.

[119] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, Feb. 1992.

[120] M. P. Wellman. A Market-Oriented Programming Environment and its Application to Distributed Multicommodity Flow Problems. *Journal of Artificial Intelligence Research*, 1:1–23, 1993.

[121] R. Wolski, J. S. Plank, J. Brevik, and T. Bryan. Analyzing Market-Based Resource Allocation Strategies for the Computational Grid. *International Journal of High Performance Computing Applications*, 15(3):258–281, Aug. 2001.

[122] L. Xiao, Y. Zhu, L. M. Ni, and Z. Xu. GridIS: An Incentive-based Grid Scheduling. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, CO, USA, Apr. 2005. IEEE Computer Society: Los Alamitos, CA, USA.

[123] V. Yarmolenko and R. Sakellariou. Towards Increased Expressiveness in Service Level Agreements. *Concurrency and Computation: Practice and Experience*, 19(14):1975–1990, 25 Sept. 2007.

[124] Y. Yemini, A. Dailianas, D. Florissi, and G. Huberman. MarketNet: Protecting Access to Information Systems through Financial Market Controls. *Decision Support Systems*, 28(1–2):205–216, Mar. 2000.

[125] C. S. Yeo and R. Buyya. Service Level Agreement based Allocation of Cluster Resources: Handling Penalty to Enhance Utility. In *Proceedings of the 7th IEEE International Conference on Cluster Computing (Cluster 2005)*, Boston, MA, USA, Sept. 2005. IEEE Computer Society: Los Alamitos, CA, USA.

[126] C. S. Yeo and R. Buyya. A Taxonomy of Market-based Resource Management Systems for Utility-driven Cluster Computing. *Software: Practice and Experience*, 36(13):1381–1419, 10 Nov. 2006.

[127] C. S. Yeo and R. Buyya. Managing Risk of Inaccurate Runtime Estimates for Deadline Constrained Job Admission Control in Clusters. In *Proceedings of the 35th International Conference on Parallel Processing (ICPP 2006)*, pages 451–458, Columbus, OH, USA, Aug. 2006. IEEE Computer Society: Los Alamitos, CA, USA.

[128] C. S. Yeo and R. Buyya. Pricing for Utility-driven Resource Management and Allocation in Clusters. *International Journal of High Performance Computing Applications*, 21(4):405–418, Nov. 2007.

[129] C. S. Yeo, R. Buyya, M. D. de Assuncao, J. Yu, A. Sulistio, S. Venugopal, and M. Placek. Utility Computing on Global Grids. In H. Bidgoli, editor, *Handbook of Computer Networks*. John Wiley and Sons, Hoboken, NJ, USA, 2007.

[130] J. Yu and R. Buyya. A Novel Architecture for Realizing Grid Workflow Using Tuple Spaces. In *Proceedings of the 5th International Workshop on Grid Computing (Grid 2004)*, pages 119–128, Pittsburgh, PA, USA, Nov. 2004. IEEE Computer Society: Los Alamitos, CA, USA.