Advance Reservation and Revenue-based Resource Management for Grid Systems

by

Anthony Sulistio

Submitted in total fulfilment of
the requirements for the degree of

Doctor of Philosophy

May 2008

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

Advance Reservation and Revenue-based Resource Management for Grid Systems

Anthony Sulistio

Supervisors: *Assoc. Prof. Rajkumar Buyya and Prof. Rao Kotagiri*

Abstract

In most Grid systems, submitted jobs are initially placed into a queue if there are no available compute nodes. Therefore, there is no guarantee as to when these jobs will be executed. This usage policy may cause a problem for time-critical applications or task graphs where jobs have inter-dependencies. To address this issue, using advance reservation (AR) in Grid systems would allow users to *secure* or *guarantee* resources prior to executing their jobs.

This thesis proposes the use of modeling and simulation, since various Grid scenarios need to be evaluated and repeated. Therefore, this thesis describes the development of GridSim, a discrete-event Grid simulation tool, which allows modeling and simulation of various properties, such as advance reservation, differentiated level of network Quality of Service (QoS), data Grid and resource discovery in a virtual organization.

This thesis investigates how AR can be incorporated and deployed in Grid systems, and determines how to increase the resource utilization. Towards accomplishing these findings, this thesis presents a system model for scheduling task graphs with advance reservation and interweaving to increase resource utilization, and proposes a new data structure, named Grid advance reservation Queue (GarQ), for administering reservations in the Grid system efficiently. In addition, this thesis provides a case for an elastic reservation model, where users can *self-select* or choose the best option in reserving their jobs, according to their QoS needs, such as deadline and budget. This thesis adapts an on-line strip packing algorithm into the elastic model to reduce the number of rejections and *fragmentations* (idle time gaps) caused by having reservations in the Grid system.

This thesis investigates how to increase resource revenue, and examines how to regulate resource supplies and reservation demands. Towards accomplishing these inquests, this thesis suggests the use of Revenue Management to determine the pricing of reservations, increase resource revenue, and regulate supply and demand. Moreover, this thesis looks into overbooking models to protect resources against unexpected cancellations and no-shows of reservations.

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

First of all, I would like to thank my principal supervisor Assoc. Prof. Rajkumar Buyya for his advice, encouragement and guidance throughout my candidature. In addition, I would like to thank my co-supervisor Prof. Rao Kotagiri for his comments and remarks. Your expertise and knowledge have influenced the direction of my research.

I am also grateful to the following people: Gokul Poduval and Assoc. Prof. Chen-Khong Tham (National University of Singapore), Prof. Dr. Wolfram Schiffmann (University of Hagen, Germany), Dr. Uros Cibej and Prof. Borut Robic (University of Ljubljana, Slovenia), Prof. Sushil Prasad (Georgia State University, USA), Agustin Caminero, Dr. Blanca Caminero, and Assoc. Prof. Carmen Carrion (Universidad de Castilla-La Mancha, Spain), and Dr. Kyong Hoon Kim (Gyeongsang National University, Korea). It has been a pleasure exchanging ideas and working with all of you.

I would like to thank Assoc. Prof. Henri Casanova (University of Hawai'i at Manoa, USA) for his excellent comments on my PhD confirmation report, and Dr. Udo Hoenig (University of Hagen, Germany) for giving me the access and technical support for the test bench structure used in my thesis. Moreover, I want to express my gratitude to Dr. Anirban Chakrabarti (Infosys Technologies, Bangalore, India), and external examiners of this thesis for their constructive comments.

I would also like to thank to all the past and current members of the GRIDS Lab, University of Melbourne. In particular, Dr. Srikumar Venugopal, Dr. Tianchi Ma, Dr. James Broberg, and Chee Shin Yeo for their help and constructive comments. My gratitude also extend to the Department's administrative and IT staff: Dr. James Bailey, Pinoo Bharucha, Cindy Sexton, Adam Hendrix, Michael Poloni, Binh Phan, and Julien Reid for their help.

Special thanks to Denzil Andrews and Donny Poh for their support and understanding. In addition, thanks to Xulio L. Albin, Jia Yu, Hussein Gibbins, Krishna Nadiminti, Prof. Dr. Christoph Reich, Matthias Banholzer, and Dr. Yoshitake Kobayashi for their company

in recreational sports and social events, which made my study more enjoyable and less stressful.

Finally, I would like to thank my family for their love, support and help in every aspect of life. I could not have done it without you.

The work presented in this thesis was partially supported by research grants from the Australian Research Council (ARC) and Australian Department of Education, Science and Training (DEST).

Anthony Sulistio
Melbourne, Australia
May 2008

Contents

List of Figures

List of Tables

List of Algorithms

List of Frequently Used Acronyms and Notations

α_p	Penalty rate, 129
API	Application Programming Interface, 13, 37
AR	Advance Reservation, 5, 13, 27, 54, 113, 146
b	Booking limit, 117, 122
$bcost$	Base cost of running a job at one time unit, 128
b_i	Booking limit for class i, 118
BoT	Bag-of-Tasks, 1, 133
$cost_{ds}$	Compensation or denied service cost, 122
C	Capacity, 118, 122
C^+	Virtual capacity, 127
CN	Compute Node, 5, 13, 37, 53, 71, 96, 146, 150
δ	A fixed time interval, 78, 100, 128, 146
dur	Reservation duration time, 97, 100, 128
DAG	Directed Acyclic Graph, 53
DCF	Denied Cost First, 129
E	Expected revenue, 118
ENR	Expected Net Revenue, 125
FCFS	First Come First Serve, 4, 40, 63, 69, 102, 146
FIFO	First In First Out, 19, 49, 57
GarQ	Grid advance reservation Queue, 9, 10, 70, 71, 94, 146
GIS	Grid Information Service, 28, 51
$IR(b_l)$	Increasing the booking limit b_l by 1, 118
$List$	A collection of task graphs and their schedules on the reserved processing elements, 57
LC-DCF	Lower Class Denied Cost First, 129
M	Number of buckets, 78, 81
mv	Maximum number of reserved compute nodes in the child nodes, 74
$maxCN$	Maximum number of compute nodes, 76, 81, 100
MI	Millions Instruction, 33

MIPS Million Instructions Per Second, 33, 46, 131

$numCN$ Number of compute nodes to be reserved, 57, 72, 96, 128

ob Overbooking limit, 121, 125
OSP On-line Strip Packing, 10, 98, 101, 110, 147

p Reservation price, 115
p_i Reservation price for class i, 118
PE Processing Element, 18, 33, 37, 53

q Show rate, 121, 125
QoS Quality of Service, 4, 13, 27, 50, 95, 96, 110, 115, 145

rv Number of reserved compute nodes, 74, 81, 146
RM Revenue Management, 7, 9, 11, 114, 147
RMS Revenue Management System, 116, 132, 147

SE Storage Element, 5, 13, 150
SL Service Level, 125
SPE Schedule Processing Element, 58
SPEC Standard Performance Evaluation Corporation, 33, 46, 131
STG Standard Task Graph, 53

τ A constant factor to differentiate reservation prices, 128
t_s Reservation start time, 57, 72, 96, 120
t_e Reservation end time, 57, 72, 96
ti_s Earliest start time interval, 97
ti_e Latest end time interval, 97
TG Task Graph, 8, 53, 57
TPE Target Processing Element, 53, 69

VO Virtual Organization, 26, 28, 50, 115, 145

x Number of bookings, 122

y Protection level, 118

Chapter 1

Introduction

This chapter presents a high-level overview of this thesis. It provides the motivation to propose advance reservation and revenue-based resource management for Grid systems. Then, it identifies the research contributions and outlines the organization of this thesis.

1.1 Grid Computing

Advances in network technologies (e.g. Web 2.0 [106]) have driven the opportunity of using network-connected computers as a single, unified computing system, known as *a cluster computer* [110]. Clusters can be used in different forms for various purposes, such as high performance computing (HPC) for more computational power than a single computer, high availability for greater reliability (in case of failure), and high throughput for longer and larger processing capability.

Grids represent a significant achievement towards the aggregation of clusters and/or other networked resources for solving large-scale data-intensive or compute-intensive applications [52]. Depending on the target application domain and purpose, Grids can be classified into several categories [156].

1. `Computational Grids.` These provide distributed computing facilities for executing compute-intensive applications, such as Monte Carlo simulations [1], and Bag-of-Tasks (BoT) applications [33], where each consists of a collection of independent tasks or jobs. Some projects such as Nimrod-G [20], SETI@home [5], and MyGrid [36]

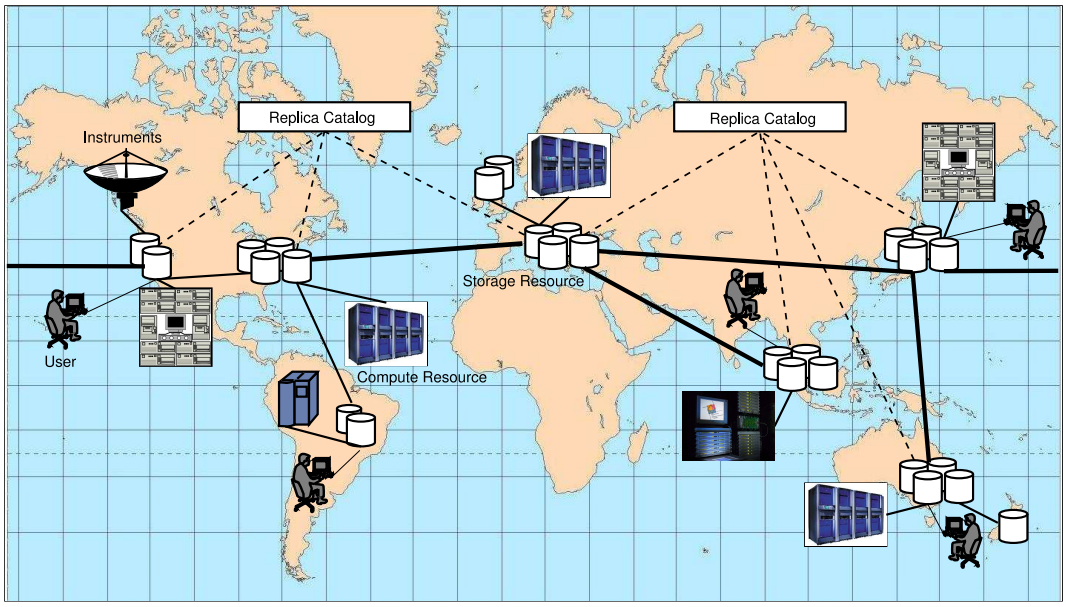


Figure 1.1: A high-level overview of Data Grid [149].

utilize Grids to schedule these applications on available resources.

2. `Data Grids.` These provide the infrastructure to access, transfer and manage large datasets stored in distributed repositories [30, 68]. In addition, data Grids focus on satisfying requirements of scientific collaborations, where there is a need for analyzing large collections of data and sharing the results. Such applications are commonly found in the area of astronomy [74], climate simulation [96], and high energy physics [68]. There are several projects involved in Data Grids, namely LHCGrid [82], Biogrid [13], Virtual Observatory [6], and Avaki EII [7].

3. `Application Service Provisioning (ASP) Grids.` These concentrate on providing access to remote applications, modules, and libraries hosted on data centers or Computational Grids, e.g. NetSolve [122].

4. `Interaction Grids.` These provide services and platforms for users to interact with each other in a real-time environment, e.g. AccessGrid [31]. Thus, this type of Grids is suitable for multimedia applications, such as video conferencing, and those that require fast networks.

5. `Knowledge Grids.` These work on knowledge acquisition, data processing, and data management. Moreover, they provide business analytics services driven by integrated

data mining services. Some projects in this field are KnowledgeGrid [25] and the EU Data Mining Grid [47].

6. `Utility Grids.` These focus on providing one or more of the above Grid services to end-users as information technology (IT) utilities on a pay-to-access basis. In addition, they set up a framework for the negotiation and establishment of contracts, and allocation of resources based on user demands. Existing projects in this area are Utility Data Center [60], at the enterprise level and Gridbus [23] at the global level.

A typical usage scenario of Grid activities, in this case for a data Grid, is shown in Figure 1.1. Scientific instruments, e.g. a satellite dish, generates large data sets which are stored in a Storage Resource. The Storage Resource then notifies a Replica Catalog (RC) about a list of available data sets. The RC acts as an indexing server for handling registrations, notifications and queries from resources and users.

Next, this RC will synchronize its information with other RCs in the Grid. When a user submits his/her jobs, a Compute Resource communicates to the nearest RC to find out the location of the required data sets (if not stored locally). Then, the Compute Resource requests to have replicas or copies of these data sets from the Storage Resource. The RCs may be arranged in different topologies depending on the requirements of the application domain, the size of the collaboration around the application and its geographical distribution [149]. Moreover, various replication techniques [137, 143, 3] may be applied to minimize the transfer time and bandwidth costs.

Based on this usage scenario, from the user's perspective, Grid computing can be considered as creating a *virtual* computer aggregating large hardware and storage infrastructures that are managed by different organizations across the world [52]. This scenario also identifies several key functionalities or components that need to be addressed by Grid resource providers:

- user interface, where users can submit and track jobs by using a command-line interface or a remote login, a graphical user interface (e.g. QMON for Sun Grid Engine [123]) or a web-based portal, such as the P-GRADE Portal [130] and the BioGrid Portal [59].

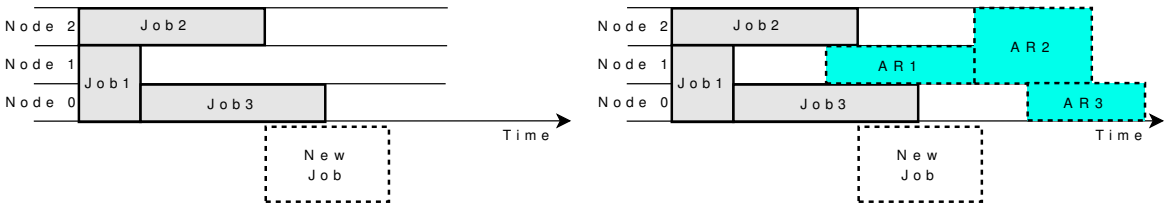
- security and access management, where users need to be authenticated and authorized before submitting jobs and using the resources respectively.
- administration and monitoring, where resource administrators can control and monitor the current state of resources, and users can track or see the progress of their jobs through an interface.
- resource discovery, where resources register their status and availability to a central server or a Replica Catalog, as shown in Figure 1.1. Thus, users can query about these resources.
- data management, where resources manage queries, replication and deletion of data sets. In addition, various replication techniques are applied.
- resource management, where resources are allocated, assigned and accessed according to Quality of Service (QoS) criteria, such as advance reservation, deadline and cost.
- job scheduling, where a local resource scheduler, such as Maui [91], executes waiting jobs in a queue based on the QoS criteria, as mentioned above.

This thesis mainly focuses on the job scheduling and resource management components of a computational Grid. This thesis aims to improve resource utilization and user satisfaction by considering novel job scheduling and reservation management strategies. This thesis also adapts an economy model to determine the pricing of each resource, increase resource revenue, and regulate supply and demand.

1.2 Motivation

1.2.1 The Need for Advance Reservation

In most Grid systems, submitted jobs are initially placed into a queue if there are no available resources by a local resource manager or scheduler. However, each Grid system may deploy a different scheduling algorithm, such as First Come First Serve (FCFS), Shortest Job First (SJF), Earliest Deadline First (EDF), or EASY Backfilling [98] that executes jobs based on different parameters, such as submission time, number of resources,



(a) without advance reservation. (b) with advance reservation.

Figure 1.2: Comparison of scheduling without and with advance reservation.

and duration of execution. Therefore, there is *no guarantee* as to when these jobs will be executed.

To address these issues and ensure that the specified resources are available for applications when required, several researchers have proposed the need for *advance reservation* (AR) [86, 132, 121]. Common resources that can be reserved or requested are compute nodes (CNs), storage elements (SEs), network bandwidth or a combination of any of those.

In general, reservation of the aforementioned resources can be categorized into two: *immediate* and *advance*. However, the main difference between these two reservations is the starting time. Immediate reservation acquires the resources to be utilized straight away, whereas advance reservation defers their usage later in the future.

Advance reservation can be useful for several applications, as described below:

- parallel applications, where each task requires multiple compute nodes simultaneously for execution.

- workflow applications, where each job may depend on the execution of other jobs in the application. Hence, it needs to wait for all of the dependencies to be satisfied before it can be executed.

- multimedia or soft real-time applications, such as video conferencing and player, where they need to have a certain amount of bandwidth to ensure a smooth broadcast of video and audio over the network. Therefore, any dropouts in a network transfer are not tolerable.

However, there are challenges in adopting advance reservation into Grids. Some of these are:

1. Significantly more complex operations and algorithms are needed for scheduling jobs, as shown in Figure 1.2. A reservation-based system needs to handle incoming bookings and queries with respect to available spaces in the current and future time, as depicted in Figure 1.2(b). Note that without AR, the future time is not considered, as illustrated in Figure 1.2(a).

2. Possibly longer waiting time for other jobs in the queue, and lower resource utilization due to fragmentations or idle time gaps, as illustrated in Figure 1.2. For example, in Figure 1.2(a), in a system without AR, a new job that requires two compute nodes can be scheduled after *Job2*. However, in a system that uses AR, this new job can only be executed after *AR2*, as depicted in Figure 1.2(b).

3. Potentiality more negotiations between the resource and users due to their requests being rejected. Hence, the system needs to manage the overheads of many requests for reserving future availability.

4. Regulating resource supplies and reservation demands during busy and non-busy periods, as this has an impact on utilization, income revenue, number of rejections and waiting time for local jobs in the system queue.

5. Possible loss of income due to cancellations and no-shows of existing reservations, since unused AR slots can not be sold to other jobs.

This thesis addresses the above challenges by modeling and scheduling of task graphs with interweaving and backfilling, using an elastic reservation model on Grid systems, and applying an economy model into these systems.

1.2.2 The Importance of Economy Model

Buyya et al. [21] introduced the Grid economy concept that provides a mechanism for regulating supply and demand, and calculates pricing policies based on these criteria. Thus, Grid economy offers an incentive for resource owners to join the Grid, and encourages users to utilize resources optimally and effectively, especially to meet the needs of critical applications.

Regulating supply and demand of resources is an important issue in AR as a study by Smith et al. [132] showed that providing AR capabilities increases waiting times of applications in the queue by up to 37% with backfilling. This study was conducted, without using any economy models, by selecting 20% of applications using reservations across different workload models. This finding implies that without economy models or any set of AR policies, a resource accepts reservations based on a first come first serve basis and subject to availability. Moreover, it also means that these reservations are treated similarly to high priority jobs in a local queue.

In order to address the above problem, Revenue Management (RM) techniques are adapted into this thesis. The main objective of RM is to maximize profits by providing the right price for every product to different customers, and periodically update the prices in response to market demands [111]. Therefore, a resource provider can apply RM techniques to *shift demands* requested by budget conscious users to off-peak periods as an example. Hence, more resources are available for users with tight deadlines in peak periods who are willing to pay more for the privilege. As a result, the resource provider gains more revenue in this scenario. So far, RM techniques have been widely adopted in various industries, such as airlines, hotels, and car rentals [92].

1.2.3 A Case for Simulation

Different scenarios need to be evaluated to ensure the effectiveness of advance reservation and revenue management techniques. Given the inherent heterogeneity of a Grid environment, it is difficult to produce performance evaluation in a *repeatable* and *controlled* manner. In addition, Grid testbeds are limited, and creating an adequately-sized testbed is expensive and time consuming. Moreover, the testbed requires the handling of different administration policies at each resource. Due to these reasons, this thesis proposes using modeling and simulation as a means of studying complex scenarios, without a full-scale implementation of Grids.

For simulating a Grid, a tool needs to be able to model the interaction of users, resource brokers (on behalf of the users), resources and the network. For these purposes, a Grid simulation tool must have at least the following functionalities:

1. Able to model heterogeneous resources, for Computational and/or Data Grids.

2. Extensible and modifiable so that various scheduling systems and economy models can be implemented and analyzed.

3. Able to store and query information about resource properties and/or data files. This can be achieved by using an indexing or catalog service.

4. Able to specify an arbitrary network topology in the simulated Grid environment.

Based on the above requirements, GridSim [22] is chosen as the preferred simulation tool. GridSim is an open-source software platform, that provides features for application composition, information services for resource discovery, and interfaces for assigning applications to resources. GridSim also has the ability to model heterogeneous computational resources of various configurations. In this thesis, a new extension is implemented to support advance reservation of nodes in compute resources.

1.3 Contributions

This thesis makes the following contributions towards research in advance reservation and revenue-based resource management for Grid systems:

1. This thesis presents a new architecture and design of GridSim in order to support advance reservation. In addition, this thesis describes the development of GridSim, which allows modeling and simulation of various properties, such as differentiated levels of network QoS [140], resource failure [24], and data Grid [139]. With the improved design and the addition of these features, GridSim offers researchers the functionality and the flexibility of simulating Grids for various types of studies, such as service-oriented computing [39], Grid meta-scheduling [3], workflow scheduling [113], and security solutions [101].

2. This thesis proposes a scheduling approach for task graphs (TGs), by using *advance reservations* to secure or guarantee resources prior to their executions. To improve resource utilization, this chapter also proposes a scheduling solution by interweav-

ing one or more TGs within the same reservation block, and backfilling with other independent jobs (if applicable).

3. This thesis puts forward a new data structure, named Grid Advance Reservation Queue (GarQ), that is tailored to handle advance reservation operations efficiently, such as searching available resources, adding, and deleting reservations. Moreover, this thesis discusses the performance of this data structure against existing ones, such as Segment Tree [17, 120], Calendar Queue [18], and Linked List [155].

4. This thesis presents an *elastic* reservation model that enables users to query resource availability with fuzzy parameters, such as duration time and number of nodes required. With this model, a resource provider can present the users with a preferred offer (a suitable AR slot) and/or a list of alternatives. Hence, the users can *self-select* or choose the best option in reserving their jobs according to their QoS constraints. Moreover, by using an on-line strip packing algorithm into the model, the model aims to reduce fragmentations or idle time gaps caused by AR, increase the number of reservations and system utilization, and minimize the waiting time of local (non-reserved) jobs in the queuing system.

5. This thesis examines how to regulate resource supplies and reservation demands. Thus, it proposes the use of Revenue Management (RM) to determine pricing of reservations in order to increase resource revenue. Hence, the resource provider can apply RM techniques to *shift demands*, and to ensure that resources are allocated to applications that are highly valued by the users. Moreover, to protect resources against unexpected cancellations and no-shows of reservations, this thesis looks into *overbooking* models that are suitable for a Grid reservation-based system. In addition, this thesis introduces several novel strategies to select which bookings or reservations to deny, based on compensation cost and user class level.

1.4 Thesis Organization

Figure 1.3 shows the organization of the rest of this thesis. In this figure, the thesis chapters are categorized into job scheduling and resource management components of a

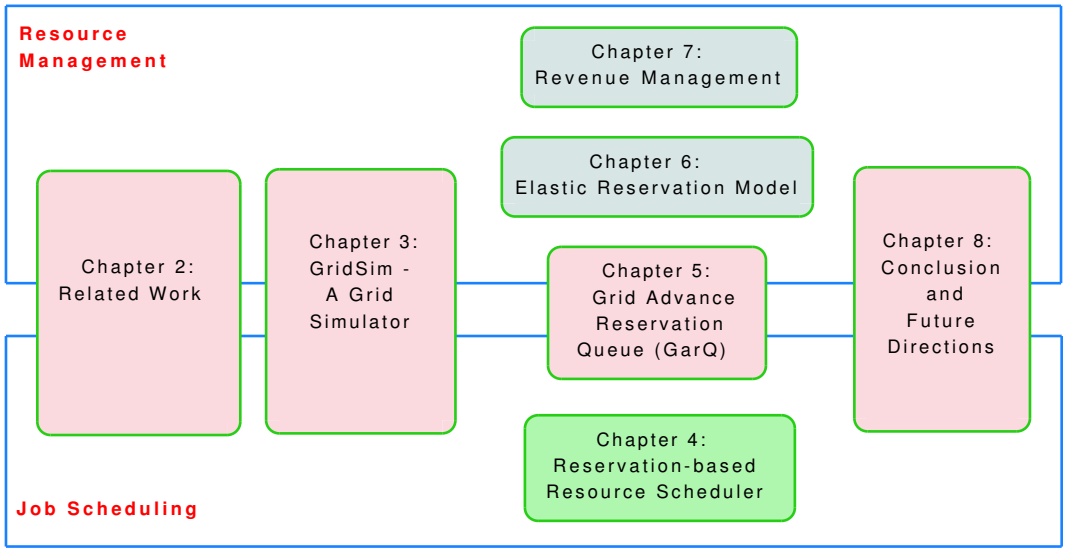


Figure 1.3: Organization of this thesis.

computational Grid, as mentioned previously.

Chapter 2 describes recent works to give an insight into the latest research advancements in projects or systems related to advance reservation in networks and Grids. Then, Chapter 3 presents a new architecture and design to GridSim to support advance reservation and other capabilities, such as differentiated levels of network QoS [140], resource failure [24], and data Grid [139]. These features of GridSim provide essential building blocks for simulating various Grid scenarios. In addition, new features can be added and incorporated easily into GridSim for the performance evaluation on topics addressed in this thesis.

Chapter 4 addresses the topic of modeling and scheduling of task graphs. This chapter proposes advance reservation to secure resources prior to their executions. In addition, to improve the resource utilization, this chapter presents a scheduling solution by interweaving one or more task graphs within the same reservation block, and backfilling with other independent jobs (if applicable).

For the next topic, this thesis introduces an elastic reservation model on Grid systems to provide users with alternative reservation slots. However, to realize this model, we need to have an efficient data structure for administering reservations. Thus, Chapter 5 presents a data structure, named a Grid advance reservation Queue (GarQ), which is built for this purpose. Then, Chapter 6 shows how GarQ is used by an On-line Strip Packing

(OSP) algorithm to find alternative offers.

Chapter 7 addresses the topic of increasing resource revenue, and regulating demand and supply. This chapter proposes the use of Revenue Management (RM) to determine pricing of reservations. In addition, this chapter introduces the concept of overbooking to protect the resource against unexpected cancellations and no-shows of reservations. Finally, Chapter 8 concludes and provides directions for future work.

These chapters are derived from various research works that have been published in various venues, detailed as follows.

- Chapter 3 is partially derived from:
  - **A. Sulistio**, U. Cibej, S. Venugopal, B. Robic and R. Buyya, A Toolkit for Modeling and Simulating Data Grids: An Extension to GridSim, *Concurrency and Computation: Practice and Experience (CCPE)*, 20(13): 1591–1609, Sep. 2008, Wiley Press, New York, USA.
  - R. Buyya and **A. Sulistio**, Service and Utility Oriented, Data Centers and Grid Computing Environments: Challenges and Opportunities for Modeling and Simulation Communities, **Keynote Paper**, In *Proceedings of the 41st Annual Simulation Symposium (ANSS'08)*, April 13–16, 2008, Ottawa, Canada.
- Chapter 4 is partially derived from:
  - **A. Sulistio**, W. Schiffmann, and R. Buyya, Advanced Reservation-based Scheduling of Task Graphs on Clusters, In *Proceedings of the 13th International Conference on High Performance Computing (HiPC'06)*, Dec. 18–21, 2006, Bangalore, India.
- Chapter 5 is partially derived from:
  - **A. Sulistio**, U. Cibej, S. Prasad, and R. Buyya, GarQ: An Efficient Scheduling Data Structure for Advance Reservations of Grid Resources, *International Journal of Parallel, Emergent and Distributed Systems (IJPEDS)*, DOI: 10.1080/17445760801988979, April 4, 2008, Taylor & Francis Publication, UK.

- Chapter 6 is partially derived from:
  - **A. Sulistio**, K. H. Kim and R. Buyya, On Incorporating an On-line Strip Packing Algorithm into Elastic Grid Reservation-based Systems, In *Proceedings of the 13th International Conference on Parallel and Distributed Systems (ICPADS'07)*, Dec. 5–7, 2007, Hsinchu, Taiwan.
- Chapter 7 is partially derived from:
  - **A. Sulistio**, K. H. Kim and R. Buyya, Using Revenue Management to Determine Pricing of Reservations, In *Proceedings of the 3rd International Conference on e-Science and Grid Computing (e-Science'07)*, Dec. 10–13, 2007, Bangalore, India.
  - **A. Sulistio**, K. H. Kim and R. Buyya, Managing Cancellations and No-shows of Reservations with Overbooking to Increase Resource Revenue, In *Proceedings of the 8th International Symposium on Cluster Computing and the Grid (CCGrid'08)*, May 19–22, 2008, Lyon, France.

Chapter 2

Related Work on Advance Reservation Projects in Networks and Grids

Advance reservation (AR) is the process of requesting resources for use at specific times in the future [132]. Common resources that can be reserved or requested are compute nodes (CNs), storage elements (SEs), network bandwidth or a combination of any of those, as mentioned earlier. This chapter describes recent works to give an insight into the latest research advancements in projects or systems related to advance reservation in networks and Grids.

2.1 Networks

Communication networks serve as a fundamental component of Grid computing, since resources are connected over public, commercial or privately-owned networks. However, without advance reservation, the network transmission quality can be degraded due to heavy demand. This may create bottlenecks for data Grids, and any other applications which primarily deal with large collections of data.

In this section, we describe reservation management systems that provide network Quality of Service (QoS) guarantee. Table 2.1 shows a summary of these works.

Table 2.1: Several systems that support advance reservation in networks.

Name	Domain	Summary
On-Demand Secure Circuits and Advance Reservation System (OSCARS) [62].	intra or single domain.	Used within the Energy Sciences Network (ESnet) [46]. It leverages technologies for enabling bandwidth reservations, such as Resource ReSerVation Protocol (RSVP) [159], Multi Protocol Label Switching (MPLS) [117], and network QoS [15].
Bandwidth Reservation for User Work (BRUW) [71].	intra or single domain.	Used within the Internet2 backbone network [73]. It allows MPLS tunnels to be dynamically created or deleted on the backbone network.
Advance Multi-domain Provisioning System (AMPS) [108].	inter or multiple federated domains.	Used as a federated reservation system within the GEANT2 network [57]. It provides for a premium Internet Protocol (IP) service or network QoS for bandwidth reservation.

2.1.1 On-Demand Secure Circuits and Advance Reservation System

On-Demand Secure Circuits and Advance Reservation System (OSCARS) [62], developed by Lawrence Berkeley National Laboratory (USA), is a prototype for enabling bandwidth reservations in a secure channel or circuit within Energy Sciences Network (ESnet) [46], a nation-wide network across the country. OSCARS aims to provide users with an easy to use and administer reservations for the whole network path. OSCARS utilizes a Reservation Manager (ReservMgr) to coordinate and configure a guaranteed bandwidth path. Therefore, users can interact with the ReservMgr through a Web-Based User Interface (WBUI) or by using the provided Application Programming Interface (API).

The ReservMgr consists of three components: the Authentication, Authorization, and Auditing Subsystem (AAAS), the Bandwidth Scheduler Subsystem (BSS), and the Path Setup Subsystem (PSS) [62]. The AAAS manages the security of OSCARS, where it authenticates username and password, digitally signs messages from network domains, allocates different resources according to users' authorizations, and logs activities related to creating or canceling reservations. The BBS schedules reservations, whereas the PSS creates and removes on-demand network paths or Label Switched Paths (LSPs) in the routers.

For the provisioning and policing of reservations, OSCARS leverages Resource ReSerVation Protocol (RSVP) [159], Multi Protocol Label Switching (MPLS) [117], and network

QoS [15]. The RSVP is used to notify the ReservMgr if the LSP can not be established due to congestion in one of the routes, whereas the MPLS is configured to establish an alternate path and label the LSP for a quick response in packet forwarding. Finally, the network QoS is used to differentiate different packets based on their Class-of-Service (CoS) attributes. Thus, packets belonging to a class with higher weight will receive a higher priority and will not be dropped in the case of network congestion.

2.1.2 Internet2 Bandwidth Reservation for User Work

Bandwidth Reservation for User Work (BRUW) [71], as part of the Internet2's Hybrid Optical and Packet Infrastructure (HOPI) project [67], is a system that allows users to reserve bandwidth over the Abilene or Internet2 backbone network [73]. The BRUW system aims to simplify the reservation process for the users, by hiding the complexity of finding the appropriate routes and network engineering tasks.

The BRUW system has three major components: user authentication, reservation verification, and reservation scheduler [71]. Initially, the users need to register and authenticate themselves to the BRUW system by using an on-line registration form. Once their applications have been approved by the system administrator, the users can request new reservations through a web portal. Then, these requests are verified against the user's privileges, the bandwidth availability, and the requested path that goes across the backbone network. If the verifications are successful, the requests are stored in the database. Finally, the resource scheduler checks the database for reservations that need to be created or deleted over MPLS tunnels on the backbone network.

2.1.3 GEANT2 Advance Multi-domain Provisioning System

The GEANT2 project [57] is a pan-European network for research and education purposes, which comprises of multiple federated domains. The Advance Multi-domain Provisioning System (AMPS) [108], as part of the GEANT2 project, is a federated reservation system for a premium Internet Protocol (IP) service. Thus, the AMPS allows users to reserve an end-to-end path on the GEANT2 network as a single request through a web portal. Note that the premium IP service is a similar term to the network QoS.

The AMPS is designed to be modular and open to future additions of premium IP networks. It has a set of loosely coupled and independent web services: Inter-domain Service (InterDS), Intra-domain Service (IntraDS), Network Information Service (NIS), and Network Element Configuration Service (NECS) [108]. The InterDS is responsible for handling users' requests and managing their reservations globally on multiple domains. In addition, the InterDS interacts with the IntraDS to make a new reservation on a local domain, and with the NIS to determine the next route of the requested end-to-end path.

Each domain on the GEANT2 network is independent. Hence, the IntraDS acts as a local resource manager and an interface to other AMPS services. The reservation request from the InterDS is first checked against local policies on available resources. Then, the IntraDS will send a notification back to the InterDS whether the request has been accepted or rejected. By having the IntraDS in each domain, networks with an existing premium IP service can participate without the need to change their existing policies.

The NIS keeps an up-to-date network information on inter- and intra-domains. Thus, it serves as a repository which handles queries from the InterDS and IntraDS about network paths and link capacities over a given period. Finally, the NECS notifies the network administrator of a local domain with an acknowledgement if a reservation has been accepted.

2.2 Grids

In this section, we present a brief description on some advance reservation projects or systems for job and resource management in Grids. Table 2.2 shows a summary of these works.

2.2.1 Maui Scheduler

Maui Scheduler [91], which was originally developed by the Maui High Performance Center (MHPC), has evolved into a community project, and is currently maintained by Cluster Resources, Inc. The Maui Scheduler is an advanced cluster scheduler that supports advance reservation, fairness, fairshare, optimization, job accounting and QoS policies, such as job prioritization, job preemption, and service access. The Maui Scheduler can act as a

Table 2.2: Some systems that support advance reservation in Grids.

Name	Resource Type	Summary
Maui Scheduler [91].	compute node.	A local job scheduler for homogeneous clusters. It is an advanced scheduler that supports fair-share, backfilling and QoS policies.
Dynamic Soft Real-Time (DSRT) Scheduling System [99, 77].	CPU. However, memory & network can be reserved through QualMan [99].	A scheduler for soft real-time applications, where resources are shared among them. The CPU broker of the DSRT system provides alternative offers for negotiation if a reservation request is rejected.
PBS Pro [102].	compute node	A local resource manager (a commercial version of PBS) with added support in advance reservation, security and information management. It can also be used to submit jobs to Globus [51].
Sun Grid Engine (SGE) [123].	compute node	An advanced resource management tool for distributed computing environments. It can interact with an external scheduler, such as Maui, for providing more comprehensive reservation functionalities.
Globus Architecture for Reservation and Allocation (GARA) [53].	network, compute node and storage.	A system that extends the Globus resource management architecture [51] to provide end-to-end QoS management for heterogeneous resources. It uses DSRT [99, 77] for reserving CPUs.
Highly-Available Resource Co-Allocator (HARC) [88].	network and compute node.	An open-source system for managing multiple reservations of various resources. It communicates with a local scheduler to determine the resource availability in the future for a particular reservation.
G-lambda Grid Scheduling System [141].	network and compute node.	A web services-based system, developed as part of the G-lambda project. It provides nodes via Globus and optical paths on a GMPLS-controlled network infrastructure.
Grid Capacity Planning [124].	compute node.	A system that provides users with reservations through negotiations, co-allocations and pricing. It uses a 3-layered negotiation protocol.

local resource manager where it has limited support for job queues and static resource partitioning to different users, groups or jobs. It can also support integration with other local resource managers, such as PBS Pro [109, 102] and Sun Grid Engine (SGE) [123], and collaboration with Grid schedulers to access resource information, job staging facilities, and advance reservations.

For the Maui Scheduler, each reservation has three major components: a set of resources, a timeframe denoting starting and ending time, and an access control list (ACL) [91]. To reserve the resources, a user needs to write a task description which contains the exact required number of attributes, such as processing elements (PEs), memory, and hard disk. The ACL specifies which users, groups or jobs can use a reservation. Then, the Maui Scheduler will find available resources based on the given task description and ACL. To improve utilization, the Maui Scheduler uses a backfilling method, which execute smaller jobs waiting later in a queue, provided that they do not affect the start time of existing reservations. The Moab Workload Manager [97] which is a commercial version of the Maui Scheduler provides the same reservation features. However, it has other advanced functionalities, such as dynamic partitioning, user statistics, fault tolerance and integration with Globus [51].

2.2.2 Dynamic Soft Real-Time (DSRT) Scheduling System

Dynamic Soft Real-Time (DSRT) scheduling system [99, 77], developed by University of Illinois at Urbana-Champaign (USA), is a reservation-based CPU management system for soft real-time (SRT) applications. SRT applications, such as in multimedia, have soft deadlines or require a minimum guarantee QoS. Thus, they are tolerable towards minor delays or lower frame rates.

In the DSRT system, resources are shared among the SRT applications. The CPU scheduler within the DSRT system is responsible for scheduling these tasks according to their reservation parameters and usage patterns (e.g. bursty or sporadic mode). Thus, it has various scheduling mechanisms, such as Periodic Constant Processing Time (PCPT), Periodic Variable Processing Time (PVPT), Aperiodic Constant Processing Utilization (ACPU) for maximum resource requirement, sustainable resource requirement, and constant resource utilization, respectively [32]. In addition, the CPU scheduler partitions the

resources to allow other non-reserved or time sharing (TS) processes to be run in parallel. However, these TS tasks are to be executed by the local operating system.

The CPU broker of the DSRT system is responsible for administering reservation requests, and performing admission tests to find out resource availability by interacting with the CPU scheduler. In addition, the CPU broker negotiates with users by providing a list of alternative offers if the original request is rejected. Finally, the CPU broker allows the users to specify what to expect in case their reservations finish early or late. In case of the reservation finishes early, the user can choose between termination and scheduling another process. In case of the reservation finishes late, the user can choose whether to allow the CPU broker to preempt or extend it for a certain period of time.

The QoS-aware Resource Management System (QualMan) [99] is an extended version of the DSRT system that reserves additional resource types, such as network and memory. Each resource type is associated with a broker and a scheduler. Thus, the SRT applications need to negotiate with different brokers individually, or they can delegate this task to the QoS broker for simplicity.

2.2.3 PBS Pro

Portable Batch System, Professional Edition (PBS Pro) [109, 102], is a local resource manager that supports scheduling of batch jobs. It is the commercial version of PBS with added features such as advance reservation, security (e.g. authentication and authorization), cycle harvesting of idle workstations, information management (e.g. up-to-date status of a resource and its queue length), and automatic input/output file staging. PBS Pro can be installed on Unix/Linux and Microsoft Windows operating systems.

PBS Pro consists of two major component types: user-level commands and system daemons or services (i.e. Job Server, Job Executor and Job Scheduler) [102]. Commands, such as submit, monitor and delete jobs, can be first submitted through a command-line interface or a graphical user interface. These commands are then processed by the Job Server service. These jobs are eventually executed by the Job Executor service or MOM. In addition, PBS Pro enables these jobs to be submitted to Globus [51] via the Globus MOM service. Finally, the Job Scheduler service enforces site policies for each job, such as job prioritization, fairshare, job distribution or load balancing, and preemption. By

default, the Job Scheduler uses the First In First Out (FIFO) approach to prioritize jobs, however, it can also use a Round Robin or fairshare approach, where jobs are ordered based on the group's usage history and resource partitions.

Reservations are treated as jobs with the highest priority by the Job Scheduler service. Hence, reservation requests need to be checked for possible conflicts with currently running jobs and existing confirmed reservations, before they are being accepted. Requests that fail this check are denied by the Job Scheduler service.

2.2.4 Sun Grid Engine (SGE)

Sun Grid Engine (SGE) is an advanced resource management tool for distributed computing environments [123]. It is deployed in a cluster and/or campus Grid testbed, where resources can have multiple owners, but they can also belong to a single site and organization. SGE enables the submission, monitoring and control of user jobs through a command line interface or a graphical user interface via QMON. In addition, SGE supports checkpointing, resource reservation, and Accounting and Reporting Console (ARCo) through a web browser.

In SGE, resources need to be registered or classified into four types of hosts. The master host controls the overall resource management activities (e.g. job queues and user access list), and runs the job scheduler. The execution host executes jobs, while the submit host is used for submitting and controlling batch jobs. Finally, the administration host is given to other hosts, apart from the master host, to perform administrative duties. By default, the master host also acts as an administration host and a submit host.

To manage resource reservations, each job is associated with a usage policy or priority, the user group, waiting time, and resource sharing entitlements [123]. Thus, the earliest available nodes will be reserved for pending jobs with higher priority by the SGE scheduler automatically. This reservation scenario is mainly needed to avoid the job starvation problem for large (parallel) jobs. On the other hand, SGE can leverage an external scheduler, such as Maui Scheduler [91] to provide more comprehensive reservation functionalities.

2.2.5 Globus Architecture for Reservation and Allocation (GARA)

Globus Architecture for Reservation and Allocation (GARA) extends the Globus resource management architecture [51], by providing advance reservations and end-to-end QoS management for heterogeneous resources, such as compute nodes, storage elements, network bandwidth or a combination of any of these [53]. GARA uses Globus toolkit's information service for resource discovery, such as obtaining site-specific policies, system characteristics (e.g. hardware architecture and network type), and its current state (e.g. availability and installed software).

GARA adopts a layered structure, where a Local Resource Allocation Manager (LRAM) provides reservation services specific to each individual resource type and a higher-level GARA External Interface (GEI) handles issues, such as registration, resource discovery, and authentication of incoming requests. To handle bandwidth reservations or network QoS, GARA uses differentiated service mechanisms (proposed by Blake et al. [15]) by implementing an expedited forwarding per-hop behavior (PHB), configuring the ingress routers that it controls, and deploying online admission control mechanisms to enable adaptive management of reservations [55]. To reserve compute nodes, GARA adopts the Dynamic Soft Real-Time (DSRT) scheduler [99] for real-time scheduling of tasks. Finally, to reserve storage elements, GARA interacts with Distributed-Parallel Storage System (DPSS) [146] to achieve high-performance data handling.

Any co-reservation or co-allocation agents can interact with GARA seamlessly, by implementing the required advance reservation and information service API or by using the Java CoG Kit package [151]. With these approaches, agents can find available resources, make the required reservations according to QoS, and submit jobs on behalf of applications or users.

2.2.6 Highly-Available Resource Co-Allocator (HARC)

Highly-Available Resource Co-Allocator (HARC) [88], developed by the Center of Computation & Technology (CCT) at Louisiana State University (USA), is an open-source system for managing multiple reservations of various resources. This can be done by users sending reservation requests to HARC via its Java API or a command-line interface. Then,

the requests are managed by HARC Acceptors. These Acceptors are responsible for interacting with an individual Resource Manager of a specific type, similar to GARA's LRAM. Next, the Resource Manager communicates with a local scheduler to determine the resource availability in the future for a particular request. Finally, the Resource Manager sends a message to users via Acceptors, whether it accepts or rejects the given reservation request. If the request is accepted, then it needs to be committed afterwards [88].

From the above description, HARC employs a two-phase commit protocol. To ensure the reliability of Acceptors and to prevent any missing messages, HARC uses Paxos Commit [61], a transaction commit protocol, where it uses multiple Acceptors for the same user to communicate with Resource Managers. With this approach, each Resource Manager will send the same message to multiple Acceptors. If the head or lead Acceptor fails, then other Acceptors will take its place automatically.

In HARC, new types of resource can be integrated easily by creating new Resource Managers. To reserve compute nodes, the HARC Compute Resource Manager works with a local batch scheduler that supports advance reservation, such as Maui Scheduler [91] or Moab Workload Manager [97]. To reserve network bandwidth, the HARC Network Resource Manager acts as a centralized scheduler that oversees the overall management of network traffic for the entire testbed [87].

2.2.7 G-lambda Grid Scheduling System

The Grid scheduling system, developed as part of the G-lambda project, is a web service system that is able to allocate resources (compute nodes and network) in advance [141]. The aim of the G-lambda project is to build a standard web service interface among resource management systems in Grid and network computing [56]. The Grid scheduling system consists of two main components: the Grid Resource Scheduler (GRS) and the Network Resource Management System (NRM).

The GRS is developed using Globus Toolkit 4 [50], a Java implementation of Web Services Resource Framework (WSRF). It handles reservation requests from applications or Grid portals. To reserve compute nodes, the GRS interacts with Computing Resource Manager (CRM) on each site. To reserve network bandwidth, the GRS communicates with Network Resource Management System (NRM). The NRM provides optical paths

on a GMPLS-controlled network infrastructure. GMPLS is a generalization of the MPLS architecture, where it supports multiple types of switching other than label switching, such as lambda and fibre (port) [89].

To satisfy the user's QoS requirements, the scheduling module inside the GRS interacts with the CRM and/or NRM to locate available reservation slots using a depth-first search scheme [141]. However, new scheduling techniques can be easily incorporated into the module without affecting the rest of the system.

2.2.8 Grid Capacity Planning

The Grid Capacity Planning system [124], developed by the University of Innsbruck (Austria), targets to provide users with reservations of Grid resource through negotiations, co-allocations and pricing. The system has a 3-layered negotiation protocol, where the allocation layer deals with reservations on a particular Grid resource, the co-allocation layer performs a selection of available nodes from all resources based on user's QoS and optimization constraints (e.g. operating system and cost of reservations), and the negotiation layer communicates with the user about suitable reservation times and their prices. However, the system only concentrates on reserving compute nodes in advance. This is done by having the allocator and co-allocator components as WSRF web services based on the Globus Toolkit 4 [50].

The allocator exists at an individual Grid site, where it uses a Vertical Split and Horizontal Shelf-Hanger (VSHSH) algorithm [124] to solve the allocation problem. In the VSHSH algorithm, nodes are dynamically partitioned into different shelves based on demands or needs. Each shelf is associated with a fixed time length, number of nodes and cost. A new reservation request is placed or offered into an adjacent shelf that is more suitable.

Then, the co-allocator collects results from allocators of various Grid sites, and produces suitable reservation slots or offers according to the user's QoS requirements. To manage different requests from the user, the co-allocator delegates these tasks to co-allocation managers (CM). Each CM manages and negotiates one user request. Thus, this approach reduces the complexities in administering reservations.

2.3 Summary

This chapter describes some recent works related to advance reservation in networks and Grids. Reserving bandwidth over optical networks can be achieved by defining a minimum bandwidth requirement. Then, the resource manager will try to establish an end-to-end communication path over inter- or intra-domain. In case of multiple or inter-domain, the resource manager will interact or negotiate with each local administrator. Thus, a user does not need to reserve network links individually.

In Grids, all the presented works are able to reserve and manage compute nodes over hetero- or homogeneous systems. This can be done by interacting with a local resource manager at each site. Few Grid systems, such as GARA [53], HARC [88] and G-lambda [141], can also reserve network bandwidth.

All the aforementioned systems are expensive and time-consuming to build, operate and maintain. Thus, these exercises may not be feasible to some researchers and students. In the next chapter, we present a Grid simulator, named GridSim. GridSim is an open-source simulator that provides comprehensive features, such as advance reservation of compute nodes, resource failure, network QoS, and simulation of data Grids. Hence, with GridSim, researchers and students can model various scenarios in networks and Grids.

Chapter 3

A Grid Simulator that Supports Advance Reservation

Often, the evaluation of complex scenarios can not feasibly be carried out on a real Grid environment due to its dynamic nature. Thus it is difficult to produce performance evaluation in a *repeatable* and *controlled* manner. In addition, Grid testbeds are limited, and creating an adequately-sized testbed is expensive and time consuming. Moreover, the testbed requires the handling of different administration policies at each resource. Therefore, it is easier to use simulation as a means of studying these complex scenarios.

This chapter presents a new extension to GridSim, a Grid simulator, to support advance reservation of compute nodes. Moreover, this chapter describes several improvements to the existing GridSim design to make it more flexible and extensible. Thus, new features can be added and incorporated easily into GridSim.

3.1 Grid Simulation Tools

Simulation has been used extensively for modeling and evaluation of real world systems, from business process and factory assembly line to computer systems design. Consequently, modeling and simulation has emerged as an important discipline and many standard and application-specific tools and technologies have been built. They include simulation languages (e.g. Simscript [128]), simulation environments (e.g. Parsec [8]), simulation li-

Table 3.1: Some recent and notable Grid simulators.

Functionalities	GridSim	OptorSim	SimGrid	MicroGrid	GangSim
Resource Extensibility	√	–	√	√	–
Data replication	√	√	–	–	–
Disk input/output overheads	√	–	–	√	–
Complex file filtering or data query	√	–	–	–	–
Scheduling user jobs	√	–	√	√	√
reservation of a resource	√	–	–	–	–
Workload trace-based simulation	√	–	√	–	√
Differentiated network QoS	√	–	–	–	–
Generate background network traffic	√	√	√	√	–
Auction framework	√	√	–	–	–

braries (e.g. SimJava2 [126]), and application specific simulators (e.g. NS-2 network simulator [103]). While there exists a large body of knowledge and tools, there are very few well-maintained tools available for application scheduling simulation in Grid computing environments. Table 2 lists some of the recent Grid simulation tools that have emerged.

OptorSim [9] is developed as part of the EU DataGrid project. It aims to mimic the structure of an EU DataGrid Project and study the effectiveness of several Grid replication strategies. It is quite a complete package as it incorporates few auction protocols and economic models for replica optimization. However, it mainly focuses more on the issue of data replication and optimization.

The SimGrid toolkit [28], developed at the University of California at San Diego (UCSD), is a C language based toolkit for the simulation of application scheduling. It supports modeling of resources that are time-shared and the load can be injected as constants or from real traces. It is a powerful system that allows creation of tasks in terms of their execution time and resources, with respect to a standard machine capability.

The MicroGrid emulator [133], undertaken at the UCSD, is modeled after Globus [51], a software toolkit used for building Grid systems. It allows execution of applications constructed using the Globus toolkit in a controlled virtual Grid resource environment. MicroGrid is actually an emulator meaning that actual application code is executed on the virtual Grid. Thus, the results produced by MicroGrid are much closer to the real world as it is a real implementation. However, using MicroGrid requires knowledge of Globus and implementation of a real system/application to study.

GangSim [43], developed at the University of Chicago, is targeted towards a study of

usage and scheduling policies in a multi-site and multi-VO (Virtual Organization) environment. It is able to combine discrete simulation techniques and modeling of real Grid components in order to achieve scalability to Grids of substantial size.

Finally, GridSim [134], with development led by the University of Melbourne, supports simulation of various types of Grids and application models scheduling. The following sections explain GridSim's capabilities, architecture, as well as the design and implementation of new extensions that have been integrated into GridSim.

3.2 GridSim Toolkit

GridSim is an open-source software platform, written in Java, that provides features for application composition, information services for resource discovery, and interfaces for assigning applications to resources. GridSim also has the ability to model the heterogeneous computational resources of various configurations [22].

By leveraging these existing functionalities, new extensions are added into GridSim to support advance reservation (AR), differentiated levels of network Quality of Service (QoS) [140], and data Grid [139]. These extensions enable GridSim to be a comprehensive tool for simulating computational and/or data Grids. Some of the GridSim features enabled by the new extensions are outlined below:

- It allows the modeling of different resource characteristics and their failure properties [24].
- It enables simulation of workload traces taken from real supercomputers.
- It supports a reservation-based mechanism for resource allocation.
- It has an auction framework, that contains several types of auction, such as English, Dutch, Double and Sealed-bid first-price auction [38].
- It allocates incoming jobs based on space- or time-shared mode.
- It has the ability to schedule compute- and/or data-intensive jobs [139].
- It provides clear and well-defined interfaces for implementing different resource allocation algorithms.

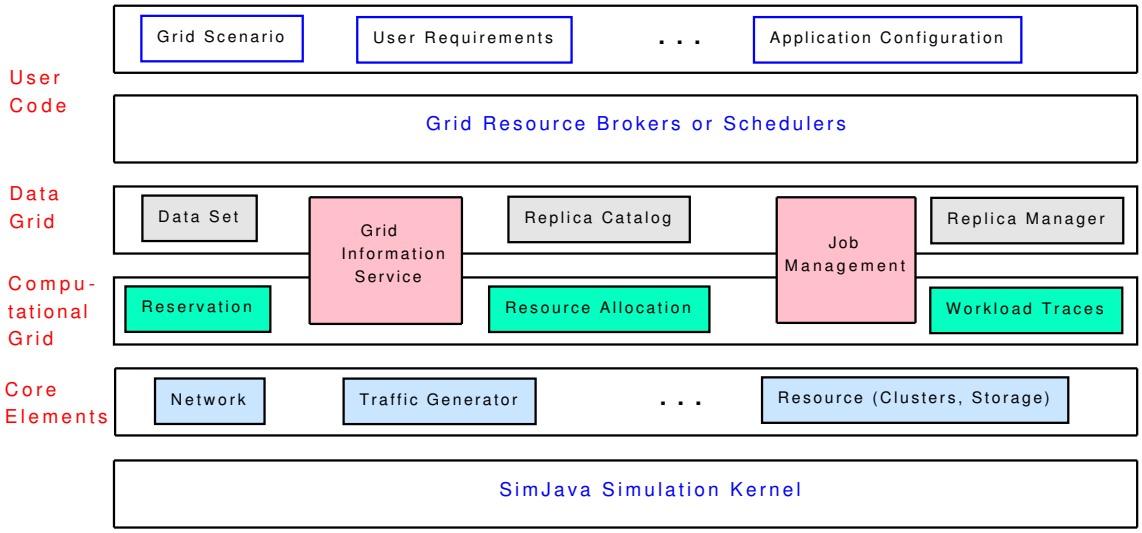


Figure 3.1: GridSim architecture

- It enables simulation of differentiated levels of network QoS [140].
- It has a background network traffic functionality based on a probabilistic distribution [140]. This is useful for simulating data-intensive jobs over a public network where the network is congested.
- It allows modeling of several regional Grid Information Service (GIS) components for resource discovery. Hence, it is able to simulate a virtual organization (VO) scenario.

In Grids, resources can be part of one or more VOs, as mentioned earlier. The concept of a VO allows users and institutions to gain access to their accumulated pool of resources to run applications from a specific field [54], such as high-energy physics or aerospace design. With these features, GridSim offers researchers the functionality and flexibility of simulating Grids for various types of studies, such as service-oriented computing [39], Grid meta-scheduling [3], workflow scheduling [113], VO-oriented resource allocation [44], and security solutions [101].

3.2.1 GridSim Architecture

The GridSim architecture with the new extensions is shown in Figure 3.1. GridSim is based on SimJava2 [126], a general purpose discrete-event simulation package implemented in Java. Therefore, the first layer at the bottom of Figure 3.1 is managed by SimJava2 for

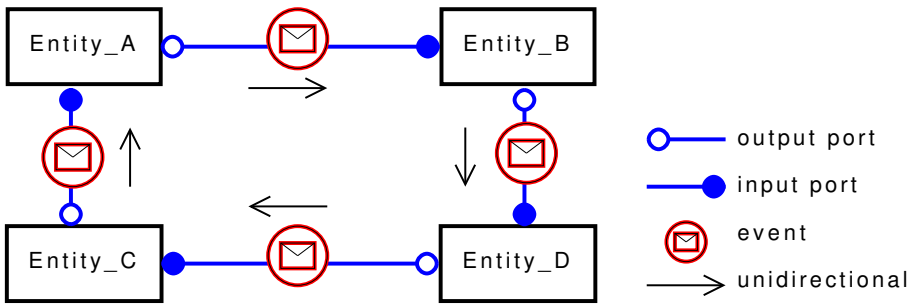


Figure 3.2: The interaction between entities in SimJava2.

handling the interaction or events among GridSim components. Also, GridSim denotes version 4.1 of the software throughout (at the time of writing this thesis).

All components in GridSim communicate with each other through message passing operations defined by SimJava2. The second layer models the core elements of the distributed infrastructure, namely Grid resources such as clusters, storage repositories and network links. These core components are absolutely essential to create simulations or experiments in GridSim.

The third and fourth layers are concerned with modeling and simulation of services specific to Computational and Data Grids respectively. Some of the services provide functions common to both types of Grids such as information about available resources and managing job submission. In case of Data Grids, job management also incorporates managing data transfers between computational and storage resources. Replica catalogs or information services for files and data, are also specifically implemented for Data Grids.

The fifth layer contains components that aid users in implementing their own schedulers and resource brokers (on behalf of users), so that they can test their own algorithms and strategies. The layer above this helps users define their own scenarios and configurations for validating their algorithms.

3.2.2 Fundamental Concepts

In SimJava2, each simulated component that interacts with others, is referred to as an *entity* [126]. The communication between entities is modeled by sending or scheduling *events* through *ports*, as shown in Figure 3.2. However, ports in SimJava2 are unidirectional communication links. For example, in this figure, *Entity_A* can only send events to

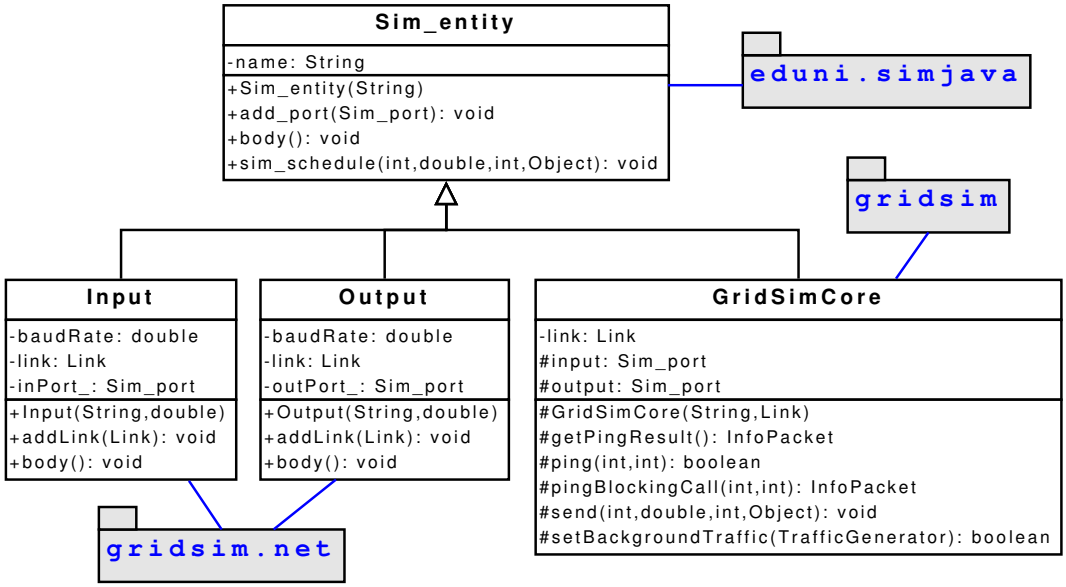


Figure 3.3: Relationship between SimJava2 and GridSim classes.

Entity_B. In addition, *Entity_A* receives events from *Entity_C* only, not from others.

An entity runs in parallel in its own thread by inheriting from the class `Sim_entity`, while its desired behavior must be implemented by overriding a `body()` method, as shown in Figure 3.3. In this figure, `Input` and `Output` are GridSim classes that are responsible for handling incoming and outgoing events through a network link respectively. Moreover, the class `GridSimCore` attaches input and output (I/O) ports and links them to another entity automatically. Thus, all lower-level implementations are hidden inside this class. In SimJava2, events and ports are represented by `Sim_event` and `Sim_port` classes respectively. Note that the class `GridSimCore` does not have the `body()` method, because its subclass will override the method for dealing with specific events. Moreover, in a class diagram (Figures 3.3, 3.5, 3.9 and 3.10) that uses Unified Modeling Language (UML) notations [112], attributes and methods are prefixed with characters +, # and − indicating access modifiers public, protected and private respectively.

To send an event, the entity needs to use either the `sim_schedule()` method of `Sim_entity` or the `send()` method of `GridSimCore`. Both methods have the same functionality, where they pass the given event into the SimJava2's simulation kernel with some important parameters, such as destination name, delay time, and tag name. The delay time refers to the waiting time of an event in the future event queue, whereas the tag name indicates a specific action or activity that needs to be performed by the receiver [126].

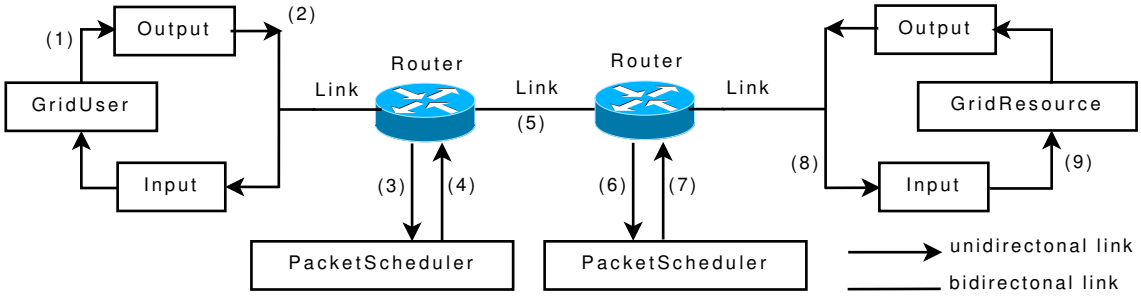


Figure 3.4: Interaction among GridSim entities in a network topology.

Figure 3.4 shows a high-level overview of the flow of communication among GridSim entities, such as *GridUser*, *Link*, *Router* and *GridResource*, all are instances of `Sim_entity`. Data sent by *GridUser* goes to its *Output* entity (step 1). The *Output* entity breaks the data into packets based on the Maximum Transmission Unit (MTU) of a network link (step 2). Then, other network components such as router and packet scheduler will deliver these packets to the destination, according to a routing table and prioritization respectively [140] (step 3–7). Finally, the data is received from a network link by *GridResource* via its *Input* entity (step 8–9). the *Input* entity assembles the packets back into the original data. Next, we briefly mention all of the GridSim packages and their functionalities.

3.2.3 New GridSim Design

Modifications or improvements to the initial GridSim design, as mentioned in [22], are needed to allow the addition of new features to be effortlessly integrated. In this section, we briefly mention some of them.

Figure 3.5 shows a class diagram hierarchy of the new GridSim design, represented by the UML notations. This figure also shows several new packages created since the initial design. However, not all classes and their complete attributes and methods are shown in this figure, as they can be found in the GridSim website [134]. The description of each GridSim package is mentioned below.

The `gridsim` package

This is the original GridSim package containing classes that form the main simulation structure of GridSim, such as

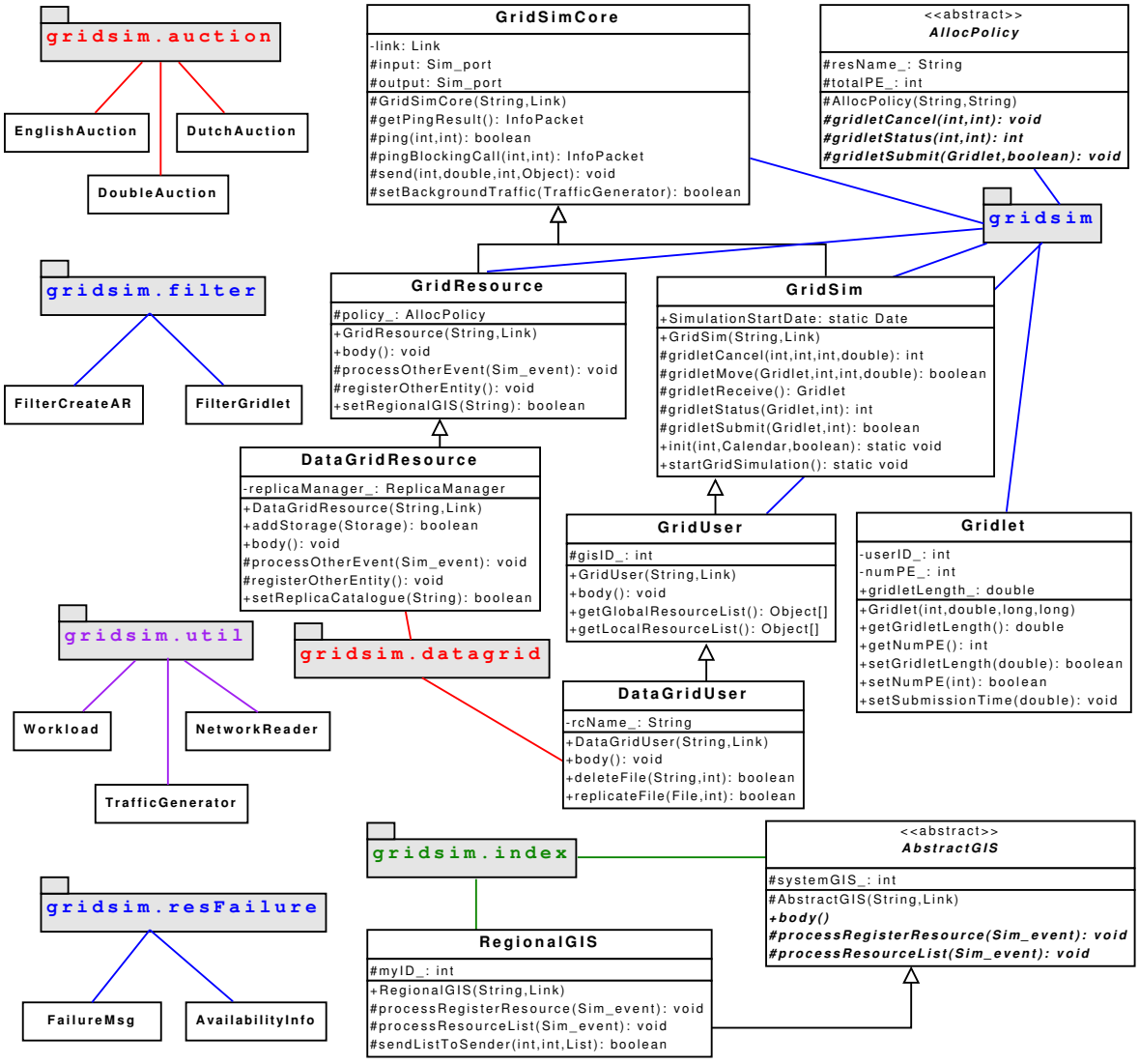


Figure 3.5: Overview of GridSim class diagram (selected classes).

- `GridSim`. This class is responsible for initialization and starting of a simulation, via `init()` and `startGridSimulation()` static methods respectively. The initialization is required in order to activate the simulation kernel of SimJava2. Moreover, it should be done before creating any of the entities.

  In the new design, this class has undergone a major change, i.e. moving all functionalities related to the I/O communications to the class `GridSimCore`, to reduce its complexities and size for easier maintenance. As a result, this class only concentrates on recording statistics and managing *gridlets* (or jobs in GridSim terms). Thus, the change makes room for new features to be added, such as allowing users to cancel, to migrate or to know the status of a particular job.

- `GridSimCore`. This base class is created, as part of the new GridSim design, in order to reduce the complexity of the class `GridSim`, as mentioned earlier. Hence, this class is mainly responsible for managing and handling the I/O communications of an entity. Moreover, with the addition of the `gridsim.net` package, an entity of this class has the ability to know the bottleneck of a network route (by using various ping methods) or to generate background network traffic in a topology (by using the class `TrafficGenerator`).

- `Gridlet`. This class represents a job package in GridSim, where it contains execution management details, such as the job length - expressed in Millions Instruction (MI), the number of processing elements (PEs) required, and the owner or user id.

- `GridUser`. This user class is created, as part of the new GridSim design, in order to communicate with a designated GIS entity (extended from the class `AbstractGIS` from the `gridsim.index` package). Hence, it allows the user to query to the GIS entity regarding to resources' availabilities and other information locally (within a VO) or globally.

- `GridResource`. This class represents a resource with various properties, such as time zone, a scheduling policy, and number of PEs and their ratings (expressed in Million Instructions Per Second (MIPS) as devised by Standard Performance Evaluation Corporation (SPEC) [135]). Therefore, resources can be modeled as

different hardware in GridSim, such as Symmetric Multi-Processing (SMP) systems or clusters.

This class has undergone a major change in the new design to allow extensibility and flexibility in creating new types of resources and scheduling algorithms. More details on this change is discussed in Section 3.3.2.

- `AllocPolicy`. This is an abstract class that handles the internal `GridResource` allocation policy. With this new design, new scheduling algorithms can be easily added into the resource entity. This can be done by extending this class and implementing the required abstract methods, as shown in Figure 3.5. More details on this change is discussed in Section 3.3.2.

This package also includes several new classes that support advance reservation, such as `ARPolicy`, `AdvanceReservation` and `ARGridResource`. These classes will be discussed in Section 3.3.

The `gridsim.auction` package

This new package contains classes that form the framework of an auction model [38] in GridSim. They include `EnglishAuction`, `DutchAuction`, and `DoubleAuction` for allocating compute nodes to the winning bidder based on English, Dutch and Double auctions respectively. Detailed explanation of this package can be found in [38].

The `gridsim.datagrid` package

This new package contains classes that form the framework of a data Grid model in GridSim. Some of them are `DataGridResource` and `DataGridUser`.

To support data Grid, a Grid resource in GridSim is associated with one or more `Storage` objects that can each model either a hard disk-based or a tape-based storage device, as shown in Figure 3.6. The resource has a `Replica Manager` which handles incoming requests for datasets located on the storage elements. In case a new replica is created, it also registers the replica with the catalog. The replica manager can be extended to incorporate different replica retention or deletion policies. A `Local Replica Catalog` object can be optionally associated with the resource to index available files

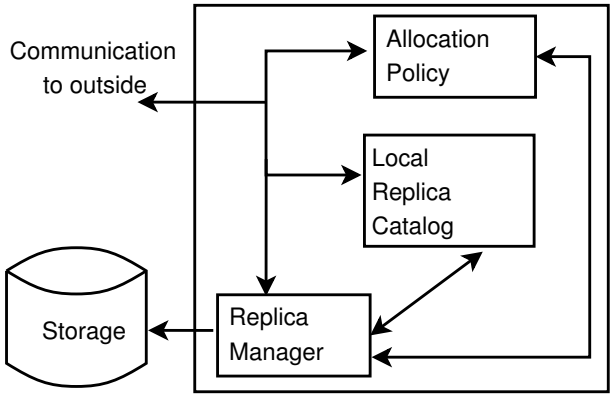


Figure 3.6: Components of a Grid resource that supports data Grid [139].

and handle direct user queries about local files. Finally, the resource has an `Allocation Policy` object which executes jobs to available compute nodes. Detailed explanation of this package can be found in [139].

The `gridsim.filter` package

This new package contains classes that form the selection of incoming events of a GridSim entity. Each class looks for a specific future event from the *Input* entity that matches certain parameters, such as tag name and sender name. For example, the class `FilterCreateAR` only finds an incoming event from a resource, regarding to creating or accepting a new reservation request. Another example, the class `FilterGridlet` looks for a specific incoming event that carries a *Gridlet* object and matches given parameters, such as resource id and user id.

The `gridsim.index` package

This new package contains classes that form the structure of multiple regional GIS entities. These classes act as an indexing service for storing a list of available resources within its regional area or from the same VO. The class `AbstractGIS` is an abstract class, which aims to provide skeletons for its child classes (e.g. `RegionalGIS`) to implement the required base functionalities of a regional GIS. In addition, the class `RegionalGIS` is able to interact with other GIS entities to find a list of resources that are located outside its VO domain.

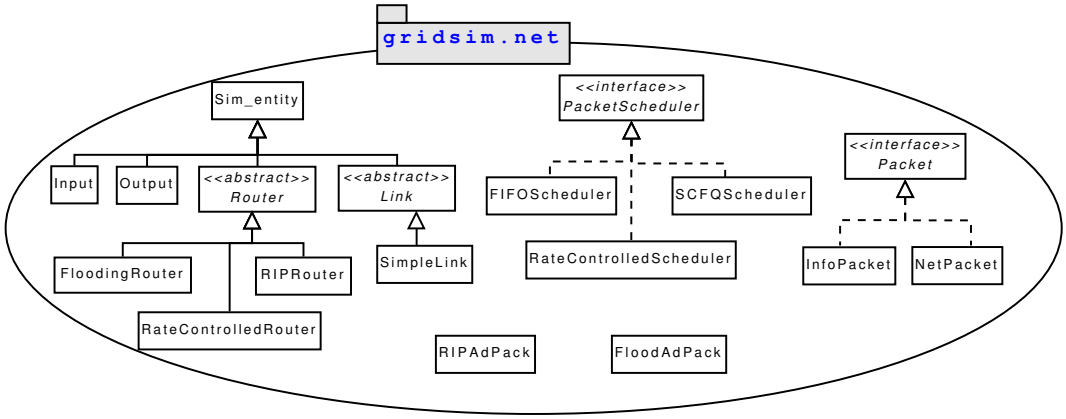


Figure 3.7: Class diagram of the gridsim.net package.

The `gridsim.net` package

This new package contains classes that form the network model [140] in GridSim, as shown in Figure 3.7. Hence, it allows GridSim entities to be connected using links and routers, with different packet scheduling policies for realistic experiments. In addition, this package enables the entity to request network information during runtime and to generate background traffic during the experiment. Detailed explanation of this package can be found in [140].

The `gridsim.resFailure` package

This new package contains classes that form the framework of resource failure and detection mechanisms [24] in GridSim. The failure models are based on probabilistic distributions with fully configurable parameters to test various scenarios. As a result, it gives GridSim a realistic model in simulating dynamic Grid computing experiments. They include `AvailabilityInfo` for storing an information about a resource availability, and `FailureMsg` for denoting a failure event of a resource. Detailed explanation of this package can be found in [24].

The `gridsim.util` package

This new package contains classes that perform other important functionalities of GridSim. Several of them are `Workload`, `TrafficGenerator`, and `NetworkReader`.

The class `Workload` is responsible for reading a workload trace file, and sending jobs to

a resource according to the trace data. The trace is recorded from a real production system. Hence, it contains several important properties (e.g. submission time and runtime), that are useful in the evaluation of resource schedulers and system utilization. The format of the trace can be in standard workload format (SWF) [49], Grid workload format (GWF) [63] or a user-defined one.

The class `TrafficGenerator` generates the inter-arrival time, packet size, and number of packets for each interval, according to various distributions that are supported by SimJava2. Some of the distributions are Bernoulli, negative exponential, and binomial. Then, these generated values are used by an *Output* entity to send background traffic packets to one or all other entities in the network topology [140].

The class `NetworkReader` has a similar functionality to `Workload`, where it parses a file and constructs a network topology automatically. Thus, this class is very useful when simulating a large topology with many network components, such as routers and links.

3.3 Design and Implementation of Advance Reservation

This section discusses the addition of advance reservation functionalities into GridSim. With this new extension, GridSim has the framework to handle:

- Creation or request of a new reservation for one or more compute nodes (CNs) or processing elements (PEs).
- Commitment of a newly-created reservation.
- Activation of a reservation once the current simulation time is the start time.
- Modification of an existing reservation.
- Cancellation and query of an existing reservation.

Note that from this chapter onwards, we use the terms PEs or CNs interchangeably.

3.3.1 States of Advance Reservation

A reservation can be in one of several states during its lifetime as shown in Figure 3.8. The life-cycle of a reservation in GridSim is influenced by recommendations from the Global

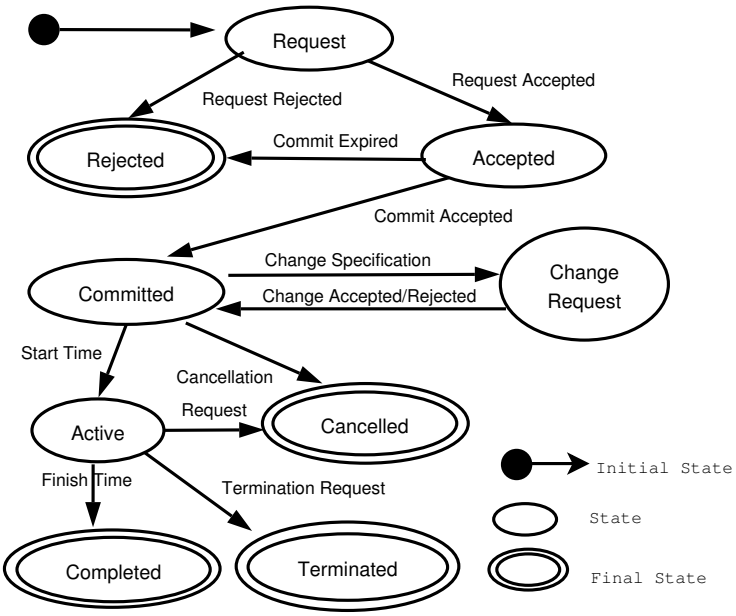


Figure 3.8: A state transition diagram for advance reservation.

Grid Forum (GGF) draft [86] and the Application Programming Interface (API) [119]. Transitions between the states are defined by the operations that a user performs on the reservation. These states are defined as follows:

- **Requested:** Initial state of the reservation, when a request for a reservation is first made.
- **Rejected:** The reservation is not successfully allocated due to full slots, or an existing reservation has expired.
- **Accepted:** A request for a new reservation has been approved.
- **Committed:** A reservation has been confirmed by a user before the expiry time, and will be honored by a resource.
- **Change Requested:** A user is trying to alter the requirements for the reservation prior to its starting. If it is successful, then the reservation is committed with the new requirements, otherwise the values remain the same.
- **Active:** The reservation's start time has been reached. The resource now executes the reservation.

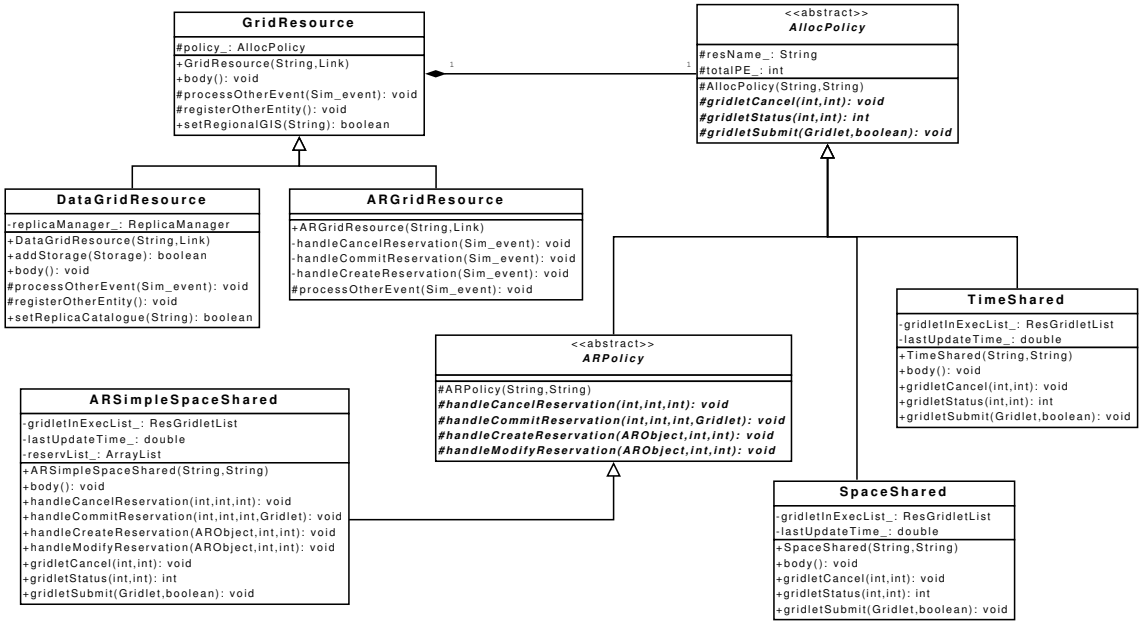


Figure 3.9: A GridSim resource class diagram (selected attributes and methods).

- **Cancelled:** A user no longer requires a reservation and requests that it is to be cancelled.
- **Completed:** The reservation's end time has been reached.
- **Terminated:** A user terminates an active reservation before the end time.

From the above states, GridSim uses a two-phase commit, where a user requests for a new reservation first. Then, if the request is accepted, then the user needs to commit the reservation within a specified time limit. If the request gets rejected, then the user needs to negotiate until successful. The following sections describe the implementation and usage of these states into GridSim.

3.3.2 Extensible Grid Resource Framework

The new GridSim design provides well-defined abstractions for configuring the resource management of a system. In GridSim, a resource is represented by a `GridResource` object. Each resource is associated with an `AllocPolicy` object that allocates computing nodes to the user jobs, depending on the given policy. Hence, the `GridResource` object, in the new GridSim design, only acts as an interface between users and the local scheduler, as shown in Figure 3.9. It is up to the scheduler to manage submitted jobs and to process

various incoming events. In contrast, the initial GridSim design, as stated in [22], puts various local schedulers and other resource functionalities into the class `GridResource`. As a result, it was hard to maintain and too complex to add new algorithms.

On the other hand, the advantage of this new design is that it gives the flexibility to implement various scheduling algorithms, such as Shortest Job First (SJF), Earliest Deadline First (EDF) and EASY Backfilling [98], as they are separate classes or entities. Hence, they are more manageable. More importantly, introducing a new scheduler into the resource does not require any modifications to an existing resource nor effect the functionalities of earlier algorithms. Currently, GridSim has `TimeShared` and `SpaceShared` objects that use Round Robin and First Come First Serve (FCFS) approaches respectively, as highlighted in Figure 3.9. Note that in this Figure, only selected attributes and methods in a class are shown.

Creating a new scheduler in the new design is as simple as extending the class `AllocPolicy` and implementing the required abstract methods, as shown in Figure 3.9. For developing algorithms that have advance reservation capabilities, they need to extend the class `ARPolicy`. For example, `ARSimpleSpaceShared` is a child of `ARPolicy` class that uses FCFS approach to schedule reserved jobs. Chapter 4 gives another example on how to schedule task graphs efficiently by using advance reservation and interweaving techniques.

The same extensibility concept is applied to creating a grid resource for different purposes. For example, `ARGridResource` is a child of the class `GridResource` that handles advance reservation operations, such as add new requests and delete existing reservations, as depicted in Figure 3.9. Another example is `DataGridResource` that extends from the class `GridResource` to manage queries or requests of various Data Grids functionalities, such as add master files or delete replicas in the system. Note that these operations or functionalities are administered in the `processOtherEvent()` method of the subclasses, where it selects an incoming event based on its tag name and refers to a private method accordingly. To register or advertise new features into a GIS entity, the subclass can override the `registerOtherEntity()` method, as shown in the class `DataGridResource` in Figure 3.9.

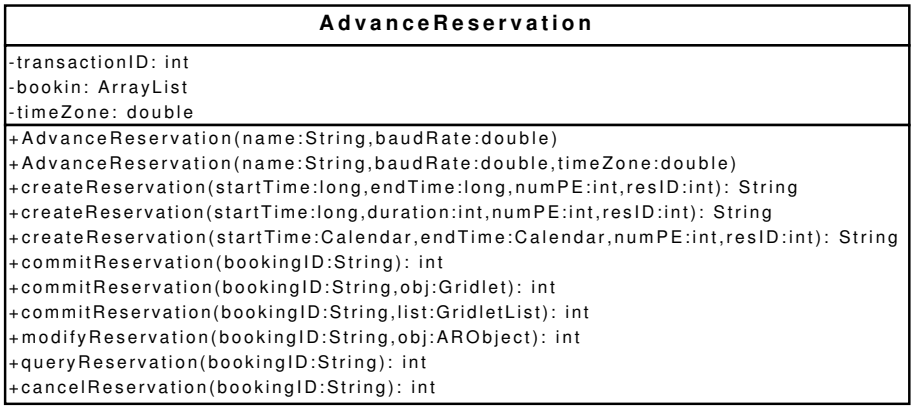


Figure 3.10: `AdvanceReservation` class diagram.

3.3.3 GridSim Application Programming Interface

The GridSim user-side API for AR is encoded in the method calls of the `AdvanceReservation` class as shown in Figure 3.10. Thus, it hides the complexity of users wanting to use the AR functionalities in GridSim. In this class diagram, attributes and methods are prefixed with characters + and − indicating access modifiers public and private respectively. However, only few methods are drawn and discussed in this chapter. Detailed API of this class can be found on the GridSim website [134]. In this section, each AR functionality is briefly discussed.

In Figure 3.10, the *transactionID* attribute is a unique identifier for a reservation, and is used to keep track of each transaction or method call associates with this reservation. Moreover, *booking* is an important attribute for storing reservations that have been accepted and/or committed. Finally, *timeZone* is another important attribute, as resources are located geographically in different time zones. Hence, a user's local time will be converted into a resource's local time when the resource receives a reservation.

For requesting a new reservation, a user needs to invoke the `createReservation()` method, as depicted in Figure 3.10. Before running a GridSim program, an initialization of some parameters is required. One of the parameters is the simulation's start time sim_t_s, where it can be a current clock time represented by a Java's `Calendar` object. Therefore, a reservation's start time needs to be ahead of sim_t_s. The start time can be of type `Calendar` object or `long` representing time in milliseconds. Reservations can also be done immediately, i.e. the current time is being used as the start time with or without

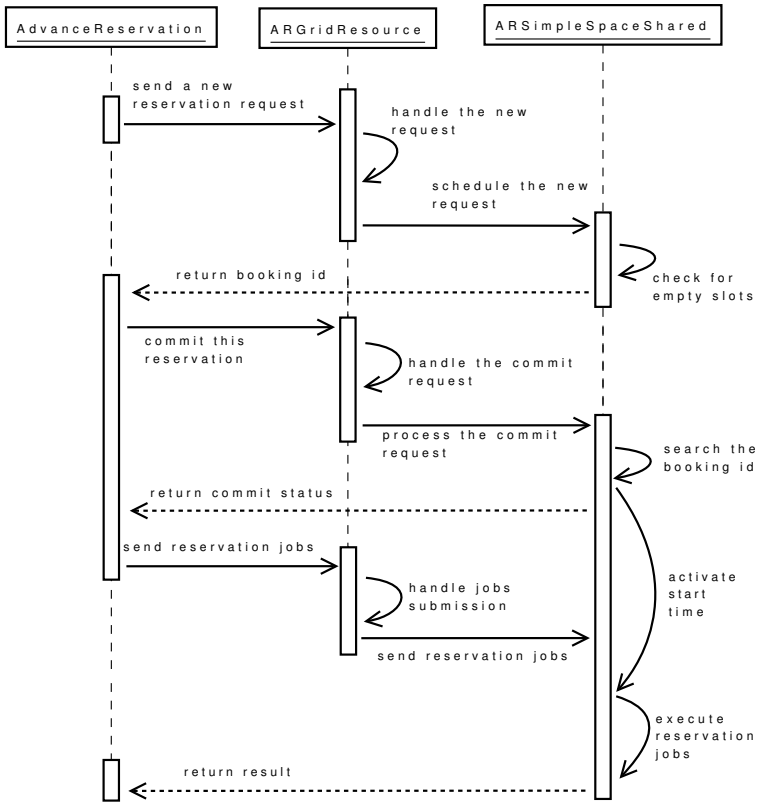


Figure 3.11: A sequence diagram for performing a new reservation in GridSim

specifying a duration time. The overall sequence from requesting a new reservation until the completion of reserved jobs, is captured in Figure 3.11.

If a new reservation has been accepted, then the `createReservation()` method will return a unique booking id, *bookingID*, as a `String` object, as shown in Figure 3.10. Otherwise, it will return an approximate busy time in the interval of 5, 10, 15, 30 and 45 in time units. The time unit can be in seconds or minutes or hours. If a request gets rejected, the user can negotiate with the resource by modifying the requirements, such as reducing the number of PEs needed or shortening the duration time, until they have come into an agreement.

Once a request for a new reservation has been accepted, the user must confirm it before the expiry time of this reservation by invoking the `commitReservation()` method. The expiry time is set by the resource or its scheduler. The `commitReservation()` method returns an integer value representing error or success code.

Committing a reservation acts as a contract for both the resource and the user. By committing, the resource is obliged to provide PEs at the specified time for a certain

period. A reservation confirmation can be done in one of the following ways:

- committing first before the expiry time by sending $bookingID$. Then, once a job is ready, committing it again with the job attached before the reservation's start time.
- committing before the expiry time together with a job. In GridSim, a job or task of an application is represented by a `Gridlet` object.
- committing before the expiry time together with a list of jobs, `GridletList`.

According to the states of AR, as shown in Figure 3.8, a reservation that has been committed successfully, can be modified before its start time. This can done by invoking the `modifyReservation()` method, which returns an integer value representing error or success code. This method has similar parameters to the `createReservation()` method, where the difference is without the need to specify a resource id $resID$. This is because $bookingID$ is unique to all resources and reservations, and it contains $resID$.

The `queryReservation()` method aims to find out the current status of the given reservation. Each reservation has one of the following status:

- `active:` the reservation has begun, and is currently being executed by a designated `GridResource` entity.
- `canceled:` the reservation has been cancelled before activation.
- `completed:` the reservation is finished, i.e. the current time is greater than the reservation's end time.
- `expired:` the reservation has passed its given expiry time before being committed.
- `not committed:` the reservation has been accepted by a resource, but not yet been committed by a user.
- `not started:` the reservation has not yet begun, i.e. the current simulation time is before the start time.
- `reservation does not exist:` the reservation's $bookingID$ does not exist or can not be found in the system.

- `terminated:` the reservation has been canceled by a user during execution or active session.

Finally, cancellation of a reservation can be done anytime before the completion time. The `cancelReservation()` method requires only *bookingID*, and returns an integer value representing error or success code. As with commitment and query of a reservation, cancellation can be done for one or more jobs.

3.4 Building a Simple Experiment with GridSim

In this section, we show some code snippets on how to build a simple experiment with GridSim. In this experiment, users are trying to reserve compute nodes to one of the resources. However, we omit input parameters on some of the class constructors and methods for simplicity. The exact input parameters and their types are listed in the GridSim API documentation at the GridSim website [134]. In addition, the GridSim website [134] provides several simple tutorial examples with detailed explanations for other GridSim functionalities.

3.4.1 Initializing GridSim

Before creating any GridSim entities and running the experiment, we need to initialize the SimJava2 simulation kernel. The initialization must be done through the `GridSim.init()` method, as shown in Listing 3.1. The method requires three parameters: the total number of users, the current calendar or the starting time of this experiment, and a flag denoting whether to record communication events among GridSim entities to a log or trace file. The trace file can be used for debugging purposes.

Listing 3.1: Code snippet for initializing GridSim.

```
1  public static void main(String[] args)
2  {
3      try {
4          int num_user = 5; // number of users created in this experiment
5          Calendar cal = Calendar.getInstance(); // experiment starting time
6          boolean trace_flag = false; // trace GridSim events or not
7          GridSim.init(num_user, cal, trace_flag);
8
9          ...  // other code for instantiating new Grid resources and users
10     }
```

```
    catch (Exception e) {
        ...  // other code for handling errors
    }
}
```

For the initialization, GridSim needs to know the total number of users in order to keep track of the number of remaining users during the simulation. As such, GridSim can notify other entities (e.g. resources and routers) about the end of simulation once all users have exited the experiment. Thus, these entities do not need to continuously wait for incoming events. GridSim also needs to know the starting time of the experiment, so users can use it to determine the reservations' start time.

3.4.2 Creating Grid Resources

Listing 3.2: Code snippet for creating a Grid resource in GridSim.

```
/**
 * Creates a GridResource entity that supports advanced reservation.
 * @param name           the resource name
 * @param totalPE        total number of processing elements (PEs) or CPUs
 * @param totalMachine   total number of machines or compute nodes
 * @param rating         the CPU speed
 */
private static ARGridResource createGridResource(String name, int totalPE,
                    int totalMachine, int rating)
{
    // Here are the steps needed to create a Grid resource:
    // 1. We need to create a list of Machines
    MachineList mList = new MachineList();
    for (int i = 0; i < totalMachine; i++) {
        // 2. A Machine contains one or more processing elements (PEs).
        PEList peList = new PEList();

        // 3. Create PEs or CPUs, and add them into the list.
        for (int k = 0; k < totalPE; k++) {
            // need to store PE id and MIPS rating (CPU speed).
            peList.add( new PE(k, rating) );
        }

        // 4. Create one Machine with its id and list of PEs or CPUs
        mList.add( new Machine(i, peList) );
    }

    // 5. Create a ResourceCharacteristics object that stores the
    //    properties of a Grid resource, e.g. operating system and time zone.
    ResourceCharacteristics resConfig = new ResourceCharacteristics(...);

    // 6. Create a network link to connect this resource
    Link link = new SimpleLink(...);

    // 7. Create a calendar that stores details about machines' availability
    ResourceCalendar cal = new ResourceCalendar(...);
```

```
37
38      // 8. Finally, we need to create a GridResource object.
39      ARGridResource gridRes;
40      try {
41          // use a scheduler that supports advance reservation
42          ARSimpleSpaceShared policy = new ARSimpleSpaceShared(...);
43
44          // then creates a grid resource entity.
45          gridRes = new ARGridResource(name, link, resConfig, cal, policy);
46      }
47      catch (Exception e) {
48          ...  // other code for handling errors
49      }
50      return gridRes;
51  }
```

The next step of building an experiment with GridSim is to create one or more Grid resources, by using the `createGridResource()` method, as shown in Listing 3.2. We first create a list of machines, where each machine has more than one PE or CPU (line 13–26). In GridSim, the total processing capability of a resource's CPU rating is modeled in the form of Million Instructions Per Second (MIPS) as devised by Standard Performance Evaluation Corporation (SPEC) [135]. In this example, we create a cluster with homogeneous machines, since they all have the same number of PE and MIPS rating.

Each resource also contains a `ResourceCharacteristics` object (line 30). This object stores static properties of a resource, such as operating system (e.g. Unix or Solaris), system architecture (e.g. Sun Ultra), and time zone. These properties may influence the users' decision in submitting their jobs. Next, we create `SimpleLink` and `ResourceCalendar` objects for linking this resource to a network and storing information about its machines' availability at various times, respectively (line 33-36). Finally, we use a scheduler that supports AR. In this case, the `ARSimpleSpaceShared` object is created (line 42).

3.4.3 Developing User's Functionalities

After creating the Grid resources, the next step is to develop the functionalities of a user in the `body()` method, as shown in Listing 3.3. For simplicity, we only highlight the important parts in this listing, i.e. make a new reservation and commit it (if accepted). Thus, we omit other details, such as how to create jobs and get the results back.

Listing 3.3: Code snippet for creating a user entity in GridSim.

```
1  /** A class that defines the behavior of a user */
```

```
2  public class UserEntity extends AdvanceReservation {
3      private int totalJob;
4      ... // other code for declaring attributes
5
6      /** A constructor */
7      public UserEntity(String name, Link link, int total) throws Exception {
8          super(name, link);
9          totalJob = total;
10         ... // other code for instantiating and initializing attributes
11     }
12
13     /** A core method that handles communications among GridSim entities */
14     public void body() {
15         // Resource Discovery for getting a list of resource IDs
16         LinkedList resList = super.getGridResourceList();
17         GridletList jobList = createGridlet(totalJob); // job creation
18
19         // Make reservation requests and send jobs to resources
20         reserveJob(resList, jobList);
21
22         ... // other code for getting the results from resources
23
24         // Signal the end of simulation for this user entity
25         super.finishSimulation();
26     }
27
28     /** A method that creates one or more Gridlets or jobs.
29      * @param total     the total number of jobs
30      */
31     private GridletList createGridlet(int total) {
32         ... // code for the creation of user jobs
33     }
34
35     /** A method that requests for a new reservation and
36      * commits the accepted reservation.
37      * @param resList   a list of resource IDs
38      * @param jobList   a list of Gridlets or jobs
39      */
40     private void reserveJob(LinkedList resList, GridletList jobList) {
41         // Want to reserve 1 day after the initial simulation time
42         Calendar cal = GridSim.getSimulationCalendar();
43         int DAY = 24 * 60 * 60 * 1000;  // in milli seconds
44         long start_time = cal.getTimeInMillis() + (1 * DAY);
45
46         // Choose a resource randomly from the list
47         Random rand = new Random();      // a random variable
48         int num = rand.nextInt(resList.size());
49         int resID = ( (Integer) resList.get(num) ).intValue();
50
51         // Determine the duration time and number of PEs required.
52         double duration = 0;     // total duration time
53         for (int i = 0; i < jobList.size(); i++) {
54             Gridlet gl = (Gridlet) jobList.get(i); // get a user job
55             num = gl.getNumPE(); // assume all jobs need the same num of PEs.
56             duration += gl.getGridletLength();     // add the duration time
57         }
58
59         // Request for a new reservation block
60         String result=super.createReservation(start_time,duration,num,resID);
61
62         ... // code for checking the result (accepted or rejected)
```

```
63
64          // If successful, commit this reservation by sending the jobs
65          int status = super.commitReservation(result, jobList);
66
67          ... // code for checking the commit result (success or failure)
68      }
69  }
```

In the `body()` method of Listing 3.3 (line 14–26), the user first needs to know about the available resources. This information can be obtained by communicating with the GIS or an indexing server (line 16). In SimJava2, each entity is associated with a unique integer ID as a means of communication with other entities. Thus, the `super.getGridResourceList()` method returns a list of resource IDs. Next, the user needs to create one or more `Gridlet` objects or jobs (line 17), before reserving the compute nodes (line 20). Finally, after the reservation has been made and the user has received the results back, the user notifies GridSim regarding to exiting the experiment, by using the `super.finishSimulation()` method (line 25). Note that in Java, the keyword `super` refers to using the method of the `UserEntity`'s parent class. In addition, we omit the description of the `createGridlet()` method (line 31–33) and how to get the results back, as the example code for these can be found on the GridSim website [134].

In the `reserveJob()` method of Listing 3.3 (line 40–68), we specify that the user wants to reserve compute nodes one day after the experiment start time (line 42–44). Then, the user randomly selects one resource from the list (line 47–49). After determining the reservation start time, the user also needs to estimate how many nodes to reserve and for how long. In this listing, we assume that all jobs need the same number of PEs (line 55). Thus, the user simply determines the duration time by adding up all of the job's length (line 52–57). Therefore, the aim of having a reservation in this example is to run batch jobs. Finally, the user sends a reservation request to the selected resource, and if it is accepted, commits the reservation straight away (line 60–65). Note that in this listing, we omit the description of the code for error checking for the case of the reservation is rejected or the commit result is unsuccessful.

3.4.4 Building a Network Topology

The next important step is to build a network topology, by linking Grid resources and users to routers, as shown in Listing 3.4. Since we use a simple topology, we can show how to set up the network entities manually in this listing. For experiments with a large network topology, the network entities can be specified in a text file. Then, GridSim builds the topology automatically by using the class `NetworkReader`, as mentioned in Section 3.2.3.

In the `connectEntity()` method of Listing 3.4, we first need to create two routers (line 13–14). Then, we attach users and Grid resources at one of these routers, by using the `attachHost()` method of the `Router` class (line 17–29). For simplicity, we choose the `FIFOScheduler` object to schedule packets on all the network links according to the First In First Out (FIFO) policy [140]. However, GridSim provides other policies for scheduling network packets, such as Self Clocked Fair Queuing (SCFQ) and a rate-jitter controlling regulator [140]. Finally, we connect the two routers altogether by using the `attachRouter()` method of the `Router` class (line 32–35).

Listing 3.4: Code snippet for linking GridSim resources and users.

```
1  /** Builds a simple network topology:
2   * User(s) ----- Router 1 ----- Router 2 ----- GridResource(s)
3   *
4   * @param resList        a list of GridResource objects
5   * @param userList       a list of UserEntity objects
6   * @param trace_flag     records network traffics in routers (true means yes)
7   */
8  public static void connectEntity(ArrayList resList, ArrayList userList,
9                                   boolean trace_flag) {
10     // Create the routers.
11     // If trace_flag is set to true, then this experiment will create
12     // the following files: router1_report.csv and router2_report.csv
13     Router r1 = new RIPRouter(``router1'', trace_flag);   // Router 1
14     Router r2 = new RIPRouter(``router2'', trace_flag);   // Router 2
15
16     // Connect all user entities with the Router 1.
17     UserEntity obj = null;
18     for (i = 0; i < userList.size(); i++) {
19         // A First In First Out (FIFO) packet scheduler will be used.
20         obj = (UserEntity) userList.get(i);
21         r1.attachHost( obj, new FIFOScheduler(...) );
22     }
23
24     // Connect all resource entities with the Router 2.
25     GridResource resObj = null;
26     for (i = 0; i < resList.size(); i++) {
27         resObj = (GridResource) resList.get(i);
28         r2.attachHost( resObj, new FIFOScheduler(...) );
29     }
30
```

```
31      // Finally, connect the two routers.
32      Link link = new SimpleLink(...);
33      FIFOScheduler r1Sched = new FIFOScheduler(...);
34      FIFOScheduler r2Sched = new FIFOScheduler(...);
35      r1.attachRouter(r2, link, r1Sched, r2Sched);
36  }
```

3.4.5 Running GridSim

The final step is to run this experiment by calling the `GridSim.startGridSimulation()` method, as shown in Listing 3.5 (line 30). This listing also highlights all the previous steps that are needed to build and run this experiment on GridSim. Once the simulation starts, the newly created entities (e.g. resources, users and routers) run in parallel in their own thread according to the runtime behavior as stated in their `body()` method.

Listing 3.5: Code snippet for building and running GridSim.

```
1  public static void main(String[] args)
2  {
3      try {
4          // Step 1: Initialize GridSim
5          int num_user = 5; // number of users created in this experiment
6          Calendar cal = Calendar.getInstance(); // experiment starting time
7          boolean trace_flag = false; // trace GridSim events or not
8          GridSim.init(num_user, cal, trace_flag);
9
10         // Step 2: Create new Grid resources
11         int total_resource = 3; // number of resources created
12         ArrayList resList = new ArrayList(total_resource);
13         for (int k = 0; k < total_resource; k++) {
14             ARGridResource res = new createGridResource(...);
15             ...  // other code for setting this resource's attributes
16             resList.add(res);
17         }
18
19         // Step 3: Create new users
20         ArrayList userList = new ArrayList(num_user);
21         for (int i = 0; i < num_user; i++) {
22             UserEntity user = new UserEntity(...);
23             ...  // other code for setting this user's attributes
24             userList.add(user);
25         }
26
27         // Step 4: Link Grid resources and users in a network topology
28         connectEntity(resList, userList, trace_flag);
29
30         // Step 5: Start the simulation
31         GridSim.startGridSimulation();
32     }
33     catch (Exception e) {
34         ...  // other code for handling errors
35     }
36 }
```

3.5 Summary

This chapter presents the development of GridSim, which allows modeling and simulation of various properties, such as differentiated level of network Quality of Service (QoS), data Grid and resource discovery in a virtual organization (VO). In addition, this chapter introduces the work done on GridSim to support advance reservation. These features of GridSim provide essential building blocks for simulating various Grid scenarios. Thus, GridSim offers researchers the functionality and flexibility of simulating Grids for various types of studies, such as service-oriented computing [39], Grid meta-scheduling [3], workflow scheduling [113], VO-oriented resource allocation [44], and security solutions [101].

To make GridSim more flexible and extensible, several improvements to the existing GridSim design were carried out. The changes include moving all functionalities related to the I/O communications in `GridSim` to a new class `GridSimCore`, creating a new class `GridUser` that allows a user to communicate to a Grid Information Service (GIS) entity, and having an abstract class `AllocPolicy` that handles the internal `GridResource` allocation policy. As a result, new features can be added and incorporated easily into GridSim for the performance evaluation on topics addressed in this thesis. These topics include modeling and scheduling of task graphs with advance reservation and interweaving, using an elastic reservation approach on Grid systems, and adapting Revenue Management techniques to determine the pricing of reservations. Thus, in the next chapter, we start by addressing the topic of modeling and scheduling of task graphs with advance reservation and interweaving.

Chapter 4

Reservation-based Resource Scheduler for Task Graphs

This chapter proposes a scheduling approach for task graphs by using advance reservation to *secure* or *guarantee* resources prior to their executions. In addition, to improve the resource utilization, this chapter also proposes a scheduling solution by interweaving one or more task graphs within the same reservation block, and backfilling with other independent jobs (if applicable).

4.1 Introduction

A Task Graph (TG) is a model of a parallel program that consists of many subtasks that can be executed simultaneously on different compute nodes (CNs) or processing elements (PEs). Subtasks exchange data via an interconnection network. The dependencies between subtasks are described by means of a Directed Acyclic Graph (DAG). Executing a TG is determined by two factors: a *node weight* that denotes the computation time of each subtask, and an *edge weight* that corresponds to the communication time between dependent subtasks [65]. Thus, to run these TGs, we need a target system that is tightly coupled by fast interconnection networks. Typically, systems such as cluster computing provides an appropriate infrastructure for running parallel programs.

Each TG can be represented in a Standard Task Graph (STG) format [65], as illustrated

```
9   3       # total subtasks and target PEs (TPEs)
0   1   0   # subtask index, node weight, and num of parents
1   1   0
2   1   1
                0    2    # parent index and edge weight
3   1   1
                0    4
4   1   2
                1    1
                2    2
5   1   1
                3    3
6   1   2
                4    2
                5    5
7   1   2
                4    1
                6    4
8   1   2               # subtask index, node weight, and num of parents
                1    5    # parent index and edge weight
                7    2    # parent index and edge weight
```

Figure 4.1: Standard Task Graph (STG) format.

in Figure 4.1. The first row of the STG format consists of two integer values, representing the total subtasks and the target PEs (TPEs) [65]. The target Processing Element (TPE) is the number of PEs required or requested by a user to execute one TG. In this figure, a TG consists of 9 subtasks ($T0 - T8$), and requires 3 TPEs. Then, a specification of individual subtask is described in a new row. Each row consists of three integers, denoting the subtask index or id, its node weight and number of parents, as shown in Figure 4.1. If the subtask has a dependency, the following row contains two numbers, specifying its parent id and the edge weight. For example, a subtask with index number 8 or $T8$ has two parents. Then, the next lines mention parents of $T8$, i.e. $T1$ and $T7$, and their edge weights of 5 and 2 time units respectively. Note that in this figure, all subtasks have a node weight of 1 time unit as an example. In addition, the STG format is similar to the one proposed by Kasahara et al. [136]. Finally, # denotes a single line comment in the STG format.

Figure 4.2 show the structure of the TG, by using the example illustrated in Figure 4.1. In this figure, a subtask's edge weight is represented by a number next to the arrow line. Scheduling the TG in a non-dedicated environment is a challenging process because of the

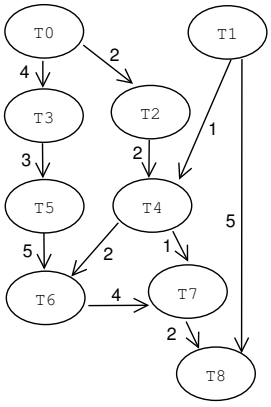


Figure 4.2: Structure of a task graph.

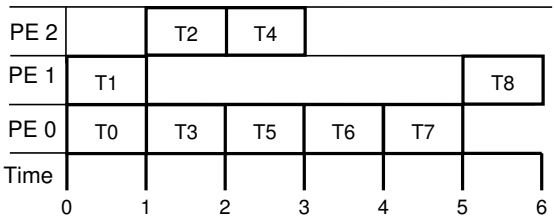


Figure 4.3: Schedule of a task graph on 3 PEs.

following constraints: *Firstly*, the TG requires a fixed number of processors for execution. Hence, a user needs to reserve the exact number of CNs. *Secondly*, due to communication overhead between the subtasks on different PEs, each subtask must be completed within a specific time period. *Finally*, each subtask needs to wait for its parent subtasks to finish executing in order to satisfy the required dependencies, as depicted in Figure 4.2. Therefore, advance reservation (AR) is needed to *secure* or *guarantee* resources prior to the execution of the subtasks.

Scheduling a TG on a resource can be visualized by a time-space diagram as shown in Figure 4.3, by using the example illustrated in Figure 4.1 and 4.2. In order to minimize the schedule length (overall computation time) and the communication costs of a TG, its subtasks must be assigned to appropriate PEs and they must be started after their parent subtasks. In this example, $T6$ depends on $T4$ and $T5$, as shown in Figure 4.2. Thus, $T6$ must wait for both subtasks to finish, and it will be scheduled on the same PE as $T5$, i.e. $PE0$, in order to minimize the communication cost. This is because executing $T6$ on $PE1$ and $PE2$ will incur a communication time of 7 and 5 time units respectively. In contrast, running $T6$ on $PE0$ after $T5$ will have a penalty of 2 time units, as shown in Figure 4.2.

If we consider DAGs with different node and edge weights, the general scheduling problem is $\mathcal{NP}$-complete [34]. Thus, in practice, heuristics are most often used to compute optimized (but not optimal) schedules, in order to minimize the total execution time. Unfortunately, the (time) optimized schedules that these algorithms produced, do not make an efficient use of the given PEs [129, 66]. In this context, the *efficiency* is measured by the ratio of the *total node weight* in relation to the *overall processing time* provided

for the TGs. As an example, in Figure 4.3, the efficiency of this TG schedule is 9/18 or 50%, which is low because $PE1$ and $PE2$ are mostly idling. If there are no idle PEs at all time, then the efficiency can be said to be optimal (100%). In Section 4.3.3, we propose a scheduling model to increase the efficiency of a task graph, by rearranging and moving subtasks, interweaving with other TGs, and backfilling with other independent jobs.

4.2 Related Work

With regards to the efficiency analysis of functional parallel programs, i.e. executing two or more tasks concurrently, there are only several works done so far. Sinnen and Sousa [129] analyze the efficiency of TG schedules, such as Economical Critical Path Fast Duplication (ECPFD) [4], Dynamic Level Scheduling (DLS) [125] and Bubble Scheduling and Allocation (BSA) [79] with respect of different Communication-to-Computation (CCR) values. The authors report that the utilization of a resource drops down if the CCR value is increased, and it also depends on the network topology of the target system. Moreover, they find that for coarse grained parallel programs (low CCR), the efficiency achieved is lower than 50%. However, it can be easily shown that this definition of efficiency is equivalent to the earlier description.

Hoenig and Schiffmann [66] also compare the efficiency of several popular heuristics, such as Dynamic Level Scheduling (DLS) [125], Earliest Time First (ETF) [70], Highest Levels First with Estimated Times (HLFET) [2] and Modified Critical-Path (MCP) [154]. They use a comprehensive test bench that is comprised of 36,000 TGs with up to 250 nodes. Essentially, it reveals that the efficiency of these schedules is mostly below 60%, which means a lot of the provided computing power is wasted. The main reason is due to the constraints of the schedule as mentioned earlier. Therefore, the main goal of our work is to increase the efficiency of these TGs by interweaving them, and backfilling with other independent jobs (if applicable).

For running DAG applications in the cluster or Grid computing environment, there are some systems available, such as Condor [144], GrADS [12], Pegasus [41], Taverna [105] and ICENI [93]. However, only ICENI provides a reservation capability in its scheduler. In comparison to our work, the scheduler inside ICENI does not consider backfilling other in-

dependent jobs with the reserved DAG applications. Hence, the ICENI resource scheduler does not consider the efficiency of the reserved applications towards resource utilization. A comprehensive survey on the characteristics and functionalities of these systems and others, is mentioned in [157].

4.3 Description of the Model

4.3.1 User Model

A user provides the following parameters during submission:

- $TG = \{T1,\ T2,\ ...,\ Tn\}$: Task Graph (TG) that consists of a set of dependent subtasks, where each subtask has one node weight and one or more edge weights. The TG is described in the STG format, as mentioned earlier.
- $List = \{TG_1,\ TG_2,\ ...,\ TG_k\}$: a collection of TGs and their schedules on the reserved PEs.
- $numCN$: number of compute nodes to be reserved.
- t_s : reservation start time.
- t_e : reservation end time.

In this model, the two-phase commit of advance reservation is applied, where a user needs to make a reservation by specifying a tuple $< numCN,\ t_s,\ t_e >$ to a resource. If the resource is not available, then the user needs to negotiate with the resource with a different time interval. Once the reservation has been accepted and confirmed, then the user sends $List$ to the resource before the start time, otherwise the reservation will be canceled. Note that the two-phase commit and states of advance reservation are explained in more details in Section 3.3.1.

4.3.2 System Model

Figure 4.4 shows the open queuing network model of a Grid system applied to our work. In this model, there are two separate queues: the AR Queue for storing reserved jobs, and

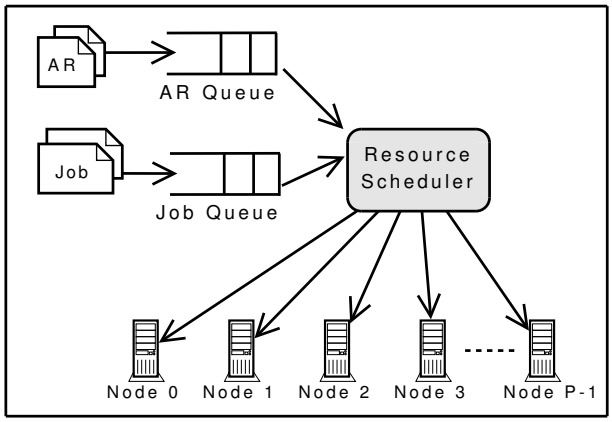
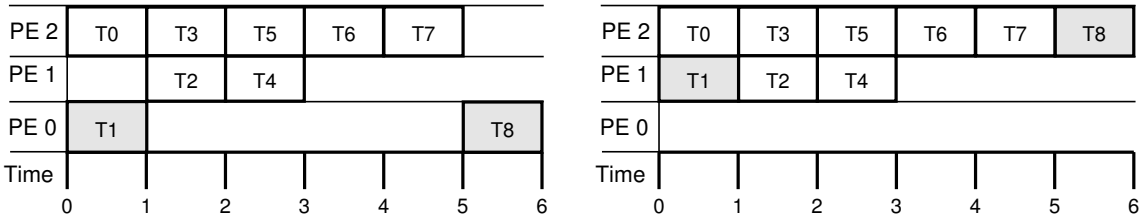


Figure 4.4: System that supports advance reservation.



(a) Rearranging subtasks.

(b) Moving subtasks.

Figure 4.5: Rearranging and moving a task graph. The shaded subtasks denote the before (a) and after (b) a moving operation.

the Job Queue for storing non-reserved jobs. The two queues have a finite buffer with size S to store objects waiting to be processed by one of P independent PEs or compute nodes. The AR Queue is a priority queue, where reserved jobs are sorted according to their reservations' start time. In contrast, the Job Queue is a queue or a First In First Out (FIFO) structure, where incoming jobs are appended to the end of the queue.

In Figure 4.4, all nodes are connected by a high-speed network. The nodes in the system can be homogeneous or heterogeneous. In this work, we assume that the system has homogeneous nodes, each having the same computing power, memory and hard disk.

In addition, the system has a Resource Scheduler, which is responsible for assigning waiting jobs in the Job Queue to available nodes. In case of reserved jobs in the AR Queue, the Resource Scheduler will schedule them according to their reservations' start time. Next, we will explain the scheduling model used by the Resource Scheduler in details.

4.3.3 Scheduling Model

In this model, we assume that we already know the optimal schedules for each TG in the AR Queue for simplicity. With this assumption, the Resource Scheduler only needs to reserve available nodes, and runs these TGs according to the given schedules. In addition, the Resource Scheduler aims to improve the average efficiency on the reserved nodes. This can be done by rearranging and moving subtasks without breaking any of the subtasks' dependencies, as shown in Figure 4.5. In the best case scenario, these methods would result in a reduction of the total number of schedule's PEs (SPEs). The schedule Processing Element (SPE) is the actual number of PEs used to execute one TG. Thus, the remaining PEs can be used to run other TG or non-reserved jobs from the Job Queue. These methods will be discussed next.

Algorithm 1: Rearranging subtasks of TG

Input: TG **and** $numCN$

```
1  index[ ] ← φ ;
2  i ← 0 ;
3  while i < numCN do
4  |   index[i].num_subtask ← get_num_subtask(TG, i);
5  |   index[i].PE_id ← i ;
6  |   i ← i + 1 ;
7  end
8  index[ ] ← sort(index[ ], NUM_SUBTASK, ASCENDING_ORDER);
9  TG ← update_schedule(index[ ], TG) ;
10 return ;
```

Rearranging Subtasks of TG

This is done by rearranging all subtasks in TG based on the total number of subtasks executed on each PE, as described in Algorithm 1. In this algorithm, we denote $index[\,]$ as an indexing array. Thus, we need to store the total number of subtasks running on each PE (line 3–7). Then, we sort $index[\,]$ from the lowest to the highest number of subtasks, where $NUM\_SUBTASK$ and $ASCENDING\_ORDER$ are constant variables (line 8). Finally, we use the $update\_schedule()$ function to update the schedules of TG (line 9), since each subtask may now be executing on a different PE.

For example, we relocate all subtasks of $PE0$, $PE1$ and $PE2$ as depicted in Figure 4.3

to $PE2$, $PE0$ and $PE1$ respectively as shown in Figure 4.5(a). This fundamental step is required as a basis for the next step.

Algorithm 2: Moving subtasks of TG to different PEs

Input: TG **and** $numCN$

```
1  PE_id[ ] ← φ;
2  for i = 0 to i < numCN do
3  |   subtask_list[ ] ← get_subtask(TG, i); // subtasks that run on the i-th PE
4  |   group_subtask(subtask_list[ ]);    // based on dependencies & edge weight
5  |   PE_id[i].add(subtask_list[ ]);    // add subtasks into list of the i-th PE
6  end
7  for i = 0 to i < numCN do
8  |   k ← i + 1;
9  |   if k ≥ numCN then
10 |   |   break;                                          // exit the loop
11 |   end
12 |   merge_subtask( PE_id[i].get_subtask(), PE_id[k].get_subtask() );
13 end
14 TG ← update_schedule(PE_id[ ], TG) ;
15 return ;
```

Moving subtasks

This is done by moving one or more subtasks from one PE to another as long as there are empty slots, as described in Algorithm 2. In this algorithm, we need to find a list of subtasks that run on a particular PE (line 3). Then, if there are two or more subtasks that depend on each other, we tag or group them as a whole (line 4). The tagging or grouping is needed to prevent them from executing into different PEs, which may incur hefty communication costs. Finally, a loop is needed to merge the two PE_id arrays into one (line 7–13), provided that there are empty slots that fit one or more subtasks.

For example, we move $T1$ and $T8$ from $PE0$, as mentioned in Figure 4.5(a), to $PE1$ and $PE2$ respectively, as shown in Figure 4.5(b). As a result, $PE0$ can be used to run another TG by interweaving, and/or backfilling with independent jobs as discussed next.

Interweaving TGs

This can be done by combining two or more TGs from $List$ and still keeping the original allocation and dependencies untouched. Algorithm 3 describes on how to interweave two

Algorithm 3: Interweaving two TGs

Input: $TG1$, $TG2$, **and** $numCN$

```
1  PE_id1[ ] ← φ;                          // storing information regarding to TG1
2  PE_id2[ ] ← φ;                          // storing information regarding to TG2
3  for i = 0 to i < numCN do
4  |  subtask_list1[ ] ← get_subtask(TG1, i);     // subtasks run on the i-th PE
5  |  subtask_list2[ ] ← get_subtask(TG2, i);
6  |  PE_id1[i].add(subtask_list1[ ]); // add subtasks into list of the i-th PE
7  |  PE_id2[i].add(subtask_list2[ ]);
8  |  PE_id1[i].start_time ← get_start_time(TG1, i);          // starting time
9  |  PE_id2[i].start_time ← get_start_time(TG2, i);
10 |  PE_id1[i].end_time ← get_end_time(TG1, i);               // ending time
11 |  PE_id2[i].end_time ← get_end_time(TG2, i);
12 end
   // check whether the given two TGs are matched for each other or not
13 result ← is_suitable(PE_id1[ ], PE_id2[ ]);
14 if result == false then
15 |  return φ ;                                  // not matched, then exit
16 end
   // determine the scheduling order of the two TGs
17 sched_first[ ] ← get_first_schedule(PE_id1[ ], PE_id2[ ]);
18 if equal(sched_first[ ], PE_id1[ ]) == true then
19 |  sched_last[ ] ← PE_id2[ ];        // TG2 is scheduled to run after TG1
20 else
21 |  sched_last[ ] ← PE_id1[ ];        // TG1 is scheduled to run after TG2
22 end
   // then sort PEs that run the TG
23 sched_first[ ] ← sort(sched_first[ ], END_TIME, DESCENDING_ORDER);
24 sched_last[ ] ← sort(sched_last[ ], START_TIME, ASCENDING_ORDER);
   // begin interweaving the two TGs
25 new_PE_id[ ] ← φ;
26 last_CN ← numCN − 1;                          // index of the last PE
27 for i = 0 to i < numCN do
28 |  gap_time ← sched_last[last_CN].start_time − sched_last[i].start_time;
29 |  sched_last[i].start_time ← sched_first[last].finish_time − gap_time;
30 |  sched_last[i] ← update_schedule(sched_last[i]);
31 |  new_PE_id[i] ← append_TG(sched_first[i], sched_last[i]);
32 end
33 new_TG ← update_schedule(new_PE_id[ ], TG1, TG2) ;
34 return new_TG ;
```

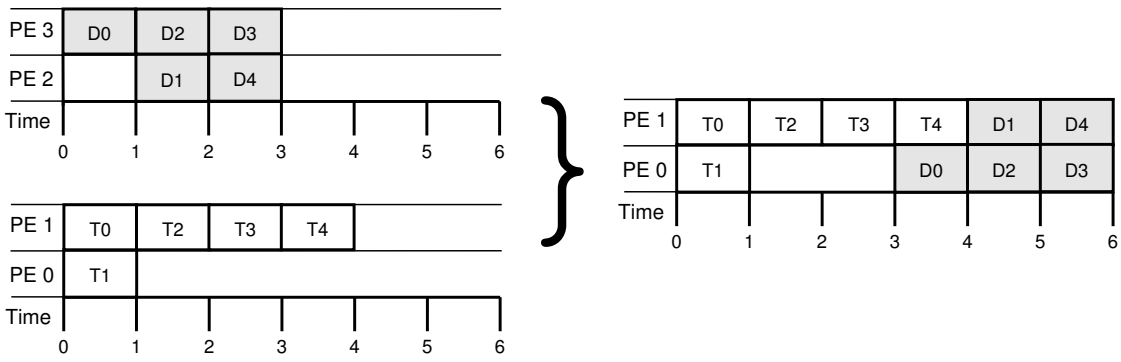


Figure 4.6: Combining the execution of two TGs by interweaving.

TGs with the use of Figure 4.6 as an example.

For each reserved PE on both TGs, as shown in Figure 4.6, we need to find a list of subtasks and its starting and ending time (Algorithm 3 line 3–12). Afterwards, we need to check whether both TGs are suitable with or matched for each other or not (line 13–16). The matching criteria need to have different starting time, ending time, or a combination of any those on one or more reserved PEs from the same TG, as shown in Figure 4.6 for example. Otherwise, the given TGs can not be interlocked properly, hence, there is no significant increase in the average efficiency of SPEs.

The next step of Algorithm 3 is to determine the scheduling order of the two TGs (line 17–22), where it also depends on the matching criteria, as mentioned earlier. For example, in Figure 4.6 on the left (with subtasks represented as D with shaded boxes), the reserved PEs for scheduling $TG1$ have the same ending time, hence, $TG1$ will be placed after $TG2$. Then, we sort the reserved PEs of each TG accordingly (line 23–24). For example, in Figure 4.6, we sort the reserved PEs of $TG1$ and $TG2$, based on the starting time in ascending order and ending time in descending order respectively. Note that in Algorithm 3, END_TIME, $DESCENDING\_ORDER$, $START\_TIME$, and $ASCENDING\_ORDER$ are constant variables.

Finally, both TGs are ready to be interweaved as one (line 25–34). This can be done by delaying or modifying the starting time of subtasks in $sched\_last[\,]$ appropriately (line 28–30). Of course this will create fragmentations or time gaps of idle processor-cycles, as depicted in Figure 4.6 on the right. However, these gaps can be hopefully closed by the following backfilling step.

Backfilling a TG or remaining gaps between interweaved TGs

This can be done if there are smaller independent jobs that can be fit in and executed, without delaying any of the subtasks of TG. Thus, we are trying to reduce fragmentations or idle time gaps. In contrast to the interweaving step, the best fitting jobs should only be selected. We start with the first gap, and look for a job that has an estimated schedule length lower or (best) equal to the gap's length. As an example, there is enough gap on $PE0$ in Figure 4.6 (on the right) to put two small independent jobs (each runs for 1 time unit) or one bigger job than needs to be scheduled for 2 time units.

4.4 Performance Evaluation

In order to evaluate the performance of our advance reservation-based scheduler (AR), we compare it with two standard algorithms, i.e. First Come First Serve (FCFS) and EASY backfilling (Backfill) [98]. We simulate the experiment with three different homogeneous target systems that consist of clusters with varying number of SPEs, i.e. 16, 32 and 64 compute nodes. Then, we run the experiment by submitting both TGs and other jobs (taken from a workload trace) into these systems.

4.4.1 Simulation Setup: Test Bench Structure

In this experiment, we use the same test bench (created by a task graph generator), as discussed in [65], to evaluate the performance of our scheduler. Therefore, we briefly describe the structure of the test bench. More detailed explanation of the test bench can be found in [65].

TGs with various properties are synthesized by a graph generator whose input parameters are varied. The directory tree that represents the structure of test bench is shown in Figure 4.7. The total number of TGs at each level within a path of the tree is shown on the right side. The parameters of a TG are described as follows (from top to bottom level in Figure 4.7):

- Graph Size (GS): denotes the number of nodes or subtasks for each TG. In Figure 4.7, the parameters of a generated TG are grouped into three categories: 7 to 12 nodes

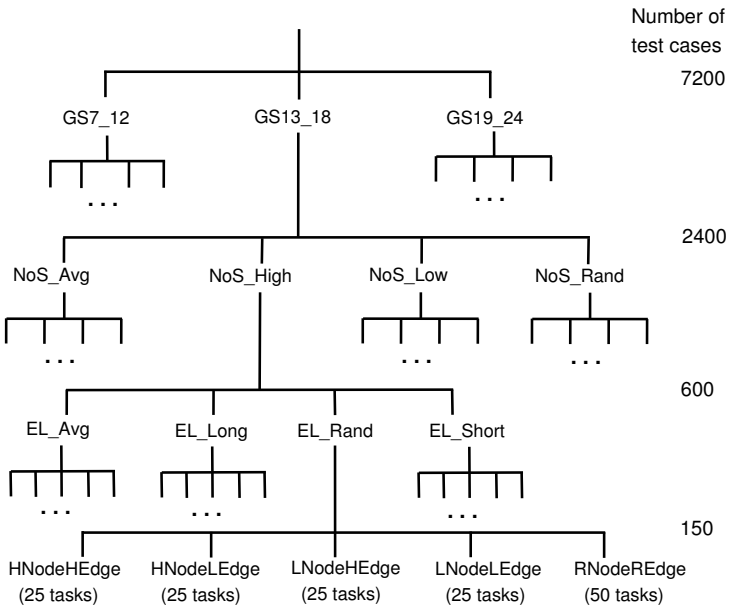


Figure 4.7: Structure of the test bench.

(GS7_12), 13 to 18 nodes (GS13_18) and 19 to 24 nodes (GS19_24).

- Meshing Degree (MD) or Number of Sons (NoS): denotes the number of dependencies between the subtasks of each TG. When a TG has a low, medium and strong meshing degree, the NoS in Figure 4.7 are NoS_Low, NoS_Avg and NoS_High respectively. TGs with random meshing degrees are represented as NoS_Rand.

- Edge Length (EL): denotes the distance between connected nodes. When a TG has a short, average & long edge length, Figure 4.7 depicts the notation as EL_Short, EL_Avg & EL_Long respectively. TGs with random edge lengths are represented as EL_Rand.

- Node- and Edge-weight: denotes the Computation-to-Communication Ratio with a combination of heavy (H), light (L) and random (R) weightings for the node & edge.

From this test bench, we also use the optimal schedules for the branches of GS7_12 and GS13_18 for both 2 and 4 TPEs. Each branch contains 2,400 task graphs, hence the maximum number of task graphs that we use is 9,600. These optimal schedules were computed and cross-checked by two independent informed search algorithms (branch-and-bound and A*) [65]. Note that at the time of conducting this experiment, the optimal

schedules of GS19_24 for 4 TPEs are not available. Therefore, in this experiment, we omit the branch of GS19_24 for both 2 and 4 TPEs.

4.4.2 Simulation Setup: Workload Trace

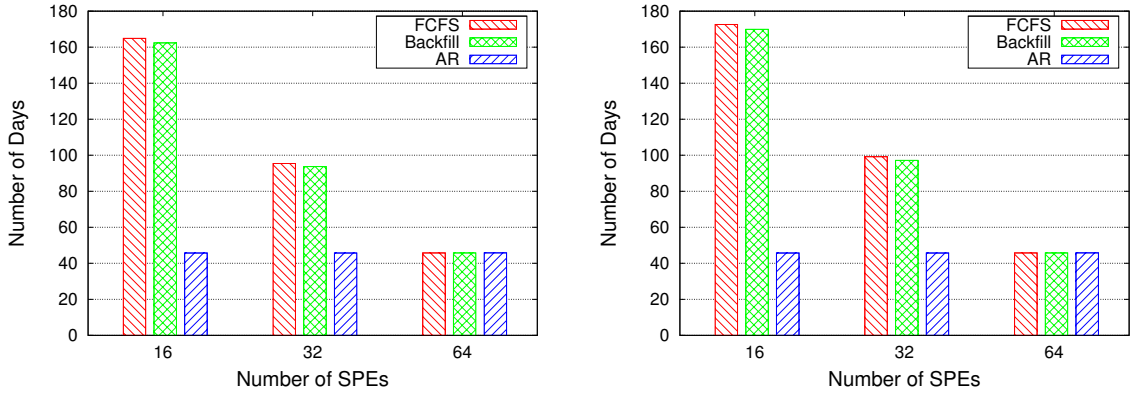
We also take two workload traces from the Parallel Workload Archive [49] for our experiment. We use the trace logs from DAS2 fs4 (Distributed ASCI Supercomputer-2 or DAS in short) cluster of Utrecht University, Netherlands and LPC (Laboratoire de Physique Corpusculaire) cluster of Universite Blaise-Pascal, Clermont-Ferrand, France. The DAS cluster has 64 CNs with 33,795 jobs, whereas the LPC cluster has 140 CNs with 244,821 jobs. The detailed analysis for DAS and LPC workload traces can be found in [84] and [94] respectively. Since both original logs recorded several months of run-time period with thousands of jobs, we limit the number of submitted jobs to be 1000, which is roughly a 5-days period from each log. If the job requires more than the total PEs of a resource, we set this job to the maximum number of PEs.

In order to submit 2,400 TGs within the 5-days period, a Poisson distribution is used. 4 TGs arrive in approximately 10 minutes for conducting the FCFS and Backfill experiments. When using the AR scheduler, we set the limit of each reservation slot to contain only 5 TGs from the same leaf of the test bench tree from Figure 4.7. Hence, only 480 reservations were created during the experiment, where every 30 minutes a new reservation is requested. If there are no available PEs, then the resource scheduler will reserve the next free ones.

4.4.3 Results

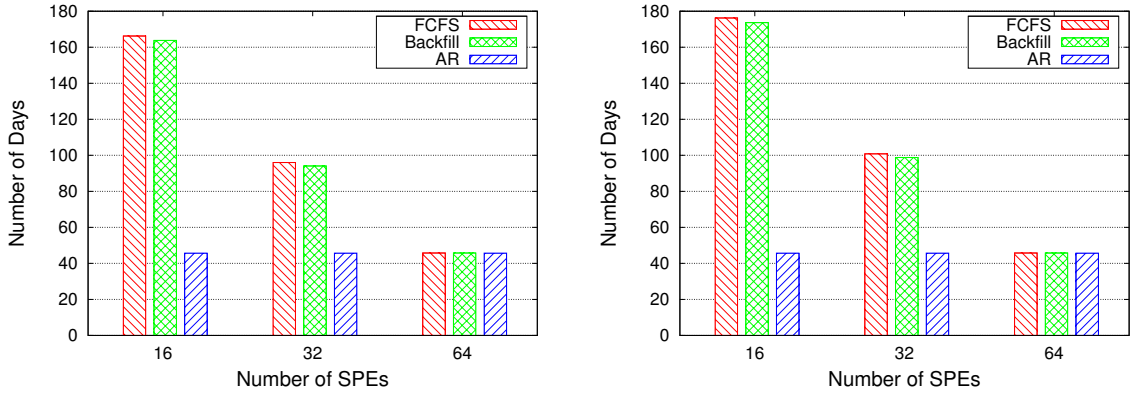
Figure 4.8 and 4.9 show the total completion time for executing TGs on the DAS trace for 2 and 4 TPEs respectively. In addition, Figure 4.10 and 4.11 show the total completion time for executing TGs on the LPC trace for 2 and 4 TPEs respectively. From these figures, the AR scheduler takes about the same amount of time to finish the TGs, regardless of the number of TPEs, SPEs, and GS branches.

From Figure 4.8 and 4.9, the AR scheduler manages to finish the experiment in 46 days (in simulation time). However, FCFS and Backfill need at least 162 and 93 days to complete the experiment for 16 and 32 SPEs respectively. For 64 SPEs, FCFS and Backfill achieve the same number of days as the AR scheduler. However, this accomplishment is



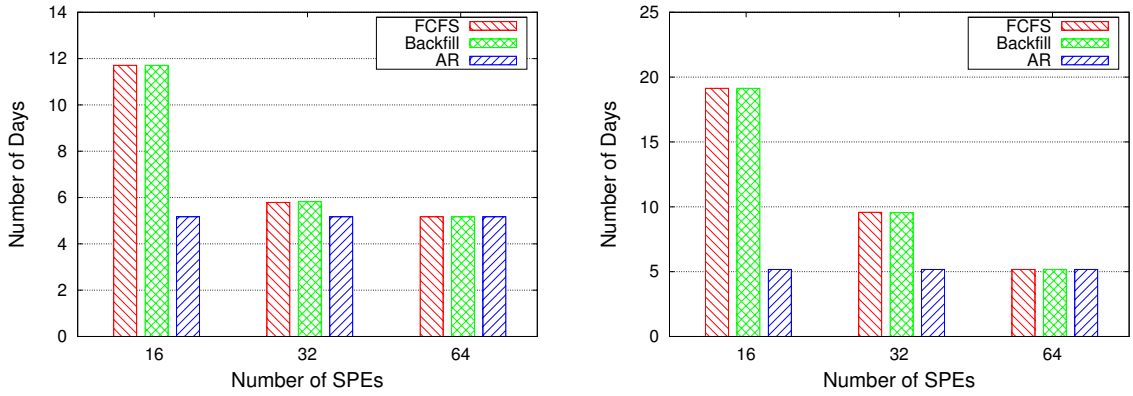
(a) Using GS7_12 branch. (b) Using GS13_18 branch.

Figure 4.8: Total completion time on the DAS trace with 2 TPEs (lower number is better).



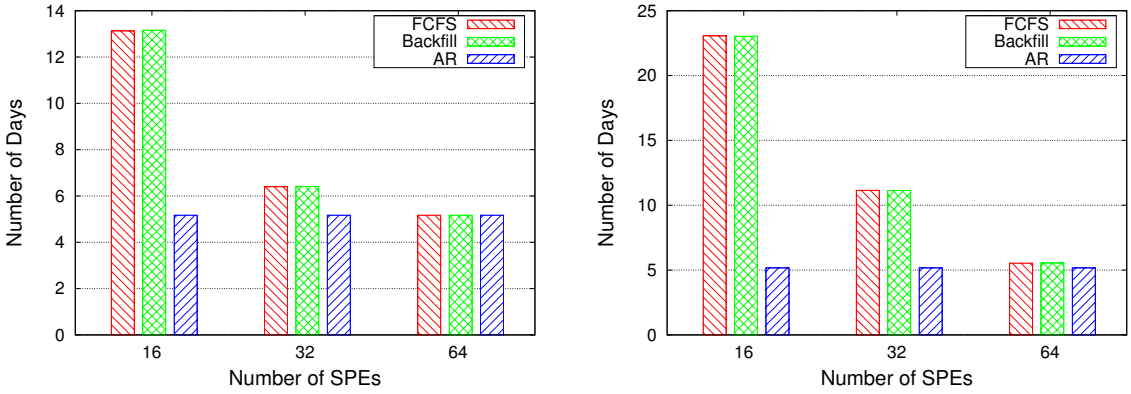
(a) Using GS7_12 branch. (b) Using GS13_18 branch.

Figure 4.9: Total completion time on the DAS trace with 4 TPEs (lower number is better).



(a) Using GS7_12 branch. (b) Using GS13_18 branch.

Figure 4.10: Total completion time on the LPC trace with 2 TPEs (lower number is better).



(a) Using GS7_12 branch.

(b) Using GS13_18 branch.

Figure 4.11: Total completion time on the LPC trace with 4 TPEs (lower number is better).

Table 4.1: Average percentage of reduction in a reservation duration time

Task Graph Parameters	2 TPEs (% reduction)			4 TPEs (% reduction)		
	GS7_12	GS13_18	Average	GS7_12	GS13_18	Average
MD Low	2.06	2.15	2.10	14.99	22.80	18.89
MD Avg	6.59	7.73	7.16	13.68	19.87	16.78
MD High	9.66	9.61	9.64	12.33	16.55	14.44
MD Rand	5.35	4.68	5.02	15.80	23.54	19.67
EL Long	0.21	0.00	0.11	9.52	11.85	10.69
EL Short	11.92	13.99	12.96	16.89	23.04	19.96
EL Avg	3.64	3.03	3.34	13.83	22.55	18.19
EL Rand	7.89	7.15	7.52	16.55	25.32	20.94
LNode LEdge	4.02	3.99	4.00	8.42	10.94	9.68
LNode HEdge	6.80	8.01	7.41	9.73	12.62	11.17
HNode LEdge	5.75	5.47	5.61	23.74	25.72	24.73
HNode HEdge	7.57	6.69	7.13	18.78	26.31	22.55
RNode REdge	5.67	6.05	5.86	12.26	24.60	18.43

Table 4.2: Average of total backfill time on the DAS trace (in seconds)

Task Graph Parameters	2 TPEs			4 TPEs		
	GS7_12	GS13_18	Average	GS7_12	GS13_18	Average
MD Low	1,089.00	432.00	760.50	711.33	209.67	460.50
MD Avg	4,499.00	2,301.33	3,400.17	2,121.33	2,585.33	2,353.33
MD High	598.67	145.00	371.83	197.67	614.33	406.00
MD Rand	943.33	1,041.67	992.50	698.67	644.33	671.50
EL Long	2,834.67	1,627.33	2,231.00	1,574.33	491.33	1,032.83
EL Short	1,811.33	1,114.00	1,462.67	467.33	2,469.33	1,468.33
EL Avg	2,263.67	379.67	1,321.67	777.33	329.00	553.17
EL Rand	220.33	799.00	509.67	910.00	764.00	837.00
LNode LEdge	1,760.67	865.33	1,313.00	981.33	329.67	655.50
LNode HEdge	602.67	74.67	338.67	436.67	9.33	223.00
HNode LEdge	620.67	102.00	361.33	201.67	146.67	174.17
HNode HEdge	1,259.67	382.00	820.83	509.33	962.67	736.00
RNode REdge	2,886.33	2,496.00	2,691.17	1,600.00	2,605.33	2,102.67

Table 4.3: Average of total backfill time on the LPC trace (in seconds)

Task Graph Parameters	2 TPEs			4 TPEs		
	GS7_12	GS13_18	Average	GS7_12	GS13_18	Average
MD Low	2,451.67	1,640.67	2,046.17	1,136.00	815.67	975.83
MD Avg	883.00	474.00	678.50	718.00	2,874.33	1,796.17
MD High	1,902.33	1,916.67	1,909.50	2,334.00	678.00	1,506.00
MD Rand	2,474.67	1,698.67	2,086.67	2,172.00	1,020.33	1,596.17
EL Long	2,018.67	1,611.33	1,815.00	1,889.00	1,419.33	1,654.17
EL Short	1,830.67	1,835.00	1,832.83	1,610.00	1,846.33	1,728.17
EL Avg	2,469.00	1,213.67	1,841.33	1,218.33	455.00	836.67
EL Rand	1,393.33	1,070.00	1,231.67	1,642.67	1,667.67	1,655.17
LNode LEdge	1,578.33	978.00	1,278.17	1,459.33	1,419.00	1,439.17
LNode HEdge	1,126.33	1,051.33	1,088.83	1,387.67	541.67	964.67
HNode LEdge	2,114.33	683.00	1,398.67	828.00	940.33	884.17
HNode HEdge	1,121.67	1,529.33	1,325.50	838.00	1,011.00	924.50
RNode REdge	1,771.00	1,488.33	1,629.67	1,847.00	1,476.33	1,661.67

mainly due to adding more nodes, rather than the effectiveness of FCFS and Backfill schedulers. The same trend can also be observed in Figure 4.10 and 4.11. From these figures, the AR scheduler manages to increase the utilization of SPEs, and minimizes the effect of having reservations in the system towards the waiting time of non-reserved jobs in the queue.

There are two main reasons that the AR scheduler manages to complete the experiments earlier than the FCFS and Backfill algorithms. The first reason is because a set of TGs in a single reservation slot can be interweaved successfully, as shown in Table 4.1.

For TGs on a GS7_12 branch for 4 TPEs, the initial reservation duration time is reduced up to 23.74% on the HNode LEdge branch. For TGs on a GS13_18 branch for 4 TPEs, the maximum reduction is 26.31% on the HNode HEdge branch. In contrast, the reduction is much smaller for 2 TPEs on the same branches. The reduction in the reservation duration time can also be referred to as an increase in the efficiency of scheduling TGs in this experiment. Overall, these results show that the achievable reduction depends on the size of the TGs and their graph properties as well.

The second reason is because there are many small independent jobs that can be used to fill in the gaps within a reservation slot, as depicted in Table 4.2 and 4.3. Hence, the AR scheduler reduces *fragmentations* or idle time gaps. However, on average, the AR scheduler manages to backfill more jobs from the LPC trace into the reservation slot compared to the DAS trace. This is due to the characteristics of workload jobs themselves. The first 1,000 jobs from the LPC trace are primarily independent jobs that require only 1 PE with an average runtime of 23.11 seconds. In contrast, the first 1,000 jobs from the DAS trace contain a mixture of independent and parallel jobs that require on average 9.15 PEs with an average runtime of 61 minutes. Thus, it explains why the total completion time on the DAS trace took much longer than the LPC one.

4.5 Summary

This chapter proposes a scheduling approach for task graphs by using advance reservation to *secure* or *guarantee* resources prior to their executions. In addition, to improve the resource utilization, this chapter also proposes a scheduling solution (AR scheduler) by interweaving one or more task graphs within the same reservation block, and backfilling with other independent jobs (if applicable).

The results show that the AR scheduler performs better than the First Come First Serve (FCFS) and EASY backfilling algorithms, in reducing both the reservation duration time and the total completion time. The AR scheduler manages to interweave a set of task graphs. Thus, it results in a reduction of the overall reservation duration time up to 23.74% and 26.31% on 7–12 nodes and 13–18 nodes, respectively, on 4 target processing elements (TPEs). However, the achievable reduction depends on the size of the task graphs and their

graph properties. Finally, the results shows that when there are many small independent jobs, the AR scheduler accomplished to fill these jobs into the reservation blocks.

Although the above findings are encouraging, there are few limitations to this approach or model. First, if there are no sufficient and suitable number of independent jobs in the queue for backfilling, then the resource utilization will suffer due to fragmentations. Second and more importantly, users must re-negotiate many times for finding available reservation slots if their earlier requests got rejected, since the resource does not provide any counter or alternative offers. Therefore, this thesis proposes an elastic reservation model to provide users with alternative reservation slots. However, to realize this model, we need to have an efficient data structure for administering reservations. Thus, in the next chapter, we present a data structure, named a Grid advance reservation Queue (GarQ), which is built for this purpose.

Chapter 5

GarQ: An Efficient Data Structure for Managing Reservations

An efficient data structure for managing reservations plays an important role in order to minimize the time required for searching available resources, adding, and deleting reservations. Therefore, this chapter proposes a new data structure, named Grid advance reservation Queue (GarQ), for administering reservations in a Grid system efficiently.

5.1 Introduction

In order to reserve available resources in a Grid system, a user must first submit a request by specifying a series of parameters such as number of compute nodes (CNs) needed, start time and duration of his/her jobs, as described in Section 3.3.1. Then, the system checks for the feasibility of this request. If there are no available nodes for the requested time period, the request is rejected. Consequently, the user may resubmit a new request with a different start time and/or duration until available nodes can be found. Given this scenario, the choice of an efficient data structure can significantly minimize the time complexity needed to search for available compute nodes, add new requests, and delete existing reservations.

Well-designed data structures provide the flexibility and easiness in implementing various algorithms, hence, some of them are tailored to specific applications. For example, a

tree-based data structure is commonly used for admission control in network bandwidth reservation [17, 153, 158]. Each tree node contains a time interval and the amount of reserved bandwidth in its subtree. Therefore, a leaf node has the smallest time interval compared to its ancestor nodes. Hence, the amount of bandwidth required for a single reservation is stored into one or more fitting nodes. In general, a tree-based structure has a time complexity of $O(log\ n)$ for searching the available bandwidth, where n is the number of tree nodes. This approach is considered to be better than using a sorted Linked-List data structure [155], which has a sequential searching method leading to $O(tot_{AR})$ time complexity, where tot_{AR} is the total number of reservations. This is because the List does not partition each reservation into a fixed time interval like a tree-based structure. Contrarily, a study done by Burchard [19] found that arrays provide better performance than a tree-based structure, such as a Segment Tree [17], for processing new requests and searching larger time intervals. The study was conducted to measure the admission speed of a bandwidth broker using each structure in a multilink admission control environment.

The previous studies are primarily focused on testing the search time of the aforementioned data structures. However, these studies do not explicitly consider *add* and *delete* operations for adding new requests and deleting existing reservations respectively, for these data structures. This is because, for reserving network bandwidth, each tree node and index in Segment Tree and Array respectively, only stores information regarding the allocated reserved bandwidth. Hence, the performance of addition and deletion can be neglected. In contrast, a data structure needs to keep additional information for reserving compute nodes in a Grid system, such as user's jobs for executing on the reserved nodes, and their status for monitoring purposes. Therefore, in order to support advance reservation in Grids, a data structure needs to perform the following basic operations:

- *search*: checking for availability of CNs in a given time interval. This operation is defined as $searchReserv(t_s,\ t_e,\ numCN)$, where t_s denotes the reservation start time, t_e denotes the reservation end time, and $numCN$ indicates the number of compute nodes to be reserved.

- *add*: inserting a new reservation request into the data structure. This operation is performed only when the previous search phase succeeded. For addition, the new

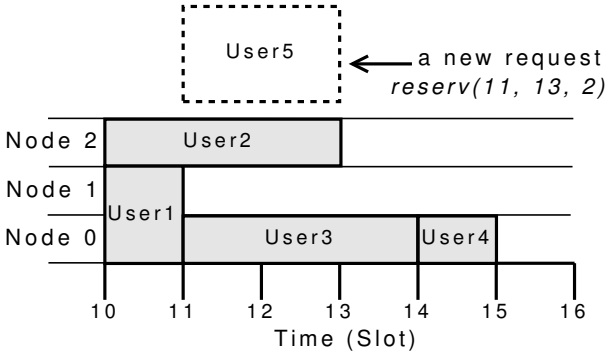


Figure 5.1: An example of advance reservations for reserving compute nodes. The maximum available compute nodes is 3. A dotted box denotes a new request.

reservation is represented as $addReserv(t_s,\ t_e,\ numCN,\ user)$, where *user* is an object storing the user's jobs and other relevant information.

- *delete*: removing the existing reservation from the data structure. This operation is conducted only when the add phase succeeds and the reservation's finish time has passed. It is described as $deleteReserv(t_s,\ t_e,\ numCN)$.

In addition, most of these studies, except done by [19], do not consider an *interval search* operation, where the data structure finds an alternative time for a rejected request. This operation helps users who requests got rejected in negotiating a suitable reservation time. Therefore, the performance of this operation also needs to be considered when choosing the appropriate data structure. This operation is represented as $suggestInterval(t_s,\ t_e,\ numCN)$.

Figure 5.1 shows an example of existing reservations represented in a time-space diagram. When a new request from *User5* arrives, the resource checks for any available CNs. In this example, the request is defined as $reserv(t_s,\ t_e,\ numCN)$, with $numCN = 2$. However, only one node is available, hence, this request will be rejected. By performing $suggestInterval(11,\ 16,\ 2)$ on this request, the system manages to find the next available time, which is from time 13 to time 15. Note that in this example, the ending time has been increased for a bigger search time range. Moreover, this *interval search* operation plays an important role in finding alternative offers in an elastic reservation model (discussed in Chapter 6).

In the next section, we describe modified versions of Linked List and Segment Tree

data structures to support *add*, *delete*, and *search*, as well as the *interval search* operation capable of dealing with advance reservations in computational Grids. For this, we had to specifically develop an algorithm for finding closest interval to a requested reservation for Segment Tree. Then, we introduce and adapt Calendar Queue [18] data structure for managing reservations. Calendar Queue is a priority queue for future event set (FES) problems in discrete event simulation. FES shares similar characteristics to advance reservations in Grids, namely it records future events, and schedules them in chronological order.

5.2 Adapting Existing Data Structures

In general, a data structure that deals with a resource reservation can be categorized into two types, i.e. a *time-slotted* and a *continuous* data structure. A time-slotted data structure divides the reservation time into fixed time intervals, also called time slots. For example, 1 slot may represent 5 minutes or 1 hour of a node's computation time. Hence, the start time and duration time of a reservation will be partitioned, compared with the existing ones and placed to the appropriate slots (if accepted). Examples of this type of data structure are Segment Tree and Calendar Queue, and they will be discussed next. In contrast, a continuous data structure, such as Linked List is more flexible, where it allows a reservation to start or finish at arbitrary times. Moreover, it obviates the need to have a minimum duration time for each reservation as compared to a time-slotted structure.

5.2.1 Segment Tree

Segment Tree, as shown in Figure 5.2, is a binary tree where each node represents a semi-open time interval $(X, Y]$. The left sibling of the node represents the interval $\left(X, \frac{X+Y}{2}\right]$, and the right sibling represents the interval $\left(\frac{X+Y}{2}, Y\right]$. Each node has also the following information:

- *rv*: the number of reserved CNs over the entire interval. When a reservation which spans the entire interval $(X, Y]$ is added, *rv* is increased by the number of CNs required by this reservation. No further descent into the child nodes is needed.

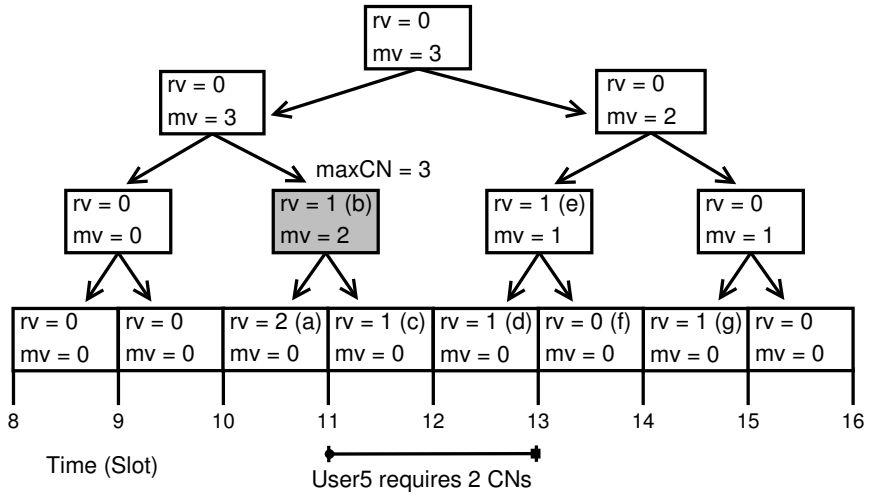


Figure 5.2: A representation of storing reservations in Segment Tree. A request from *User5* is rejected, because node (b), representing a time interval (10, 12], uses 2 nodes already.

- mv: the maximum number of reserved CNs in the child nodes. In the leaf nodes, the mv value is 0. The total number of reserved CNs in the interval of a leaf node is the sum of all rv of nodes on the path from the root node to the leaf node.

An example of a Segment Tree is shown in Figure 5.2, where it uses the same example as in Figure 5.1. Note that the complete tree in Figure 5.2 is not drawn here due to lack of space. However, the height of Segment Tree can be computed as:

$$height = \log_2 \left(\frac{interval\_length}{slot\_length} \right) \tag{5.1}$$

where $interval\_length$ is the length of the whole interval we want to cover, and $slot\_length$ is the length of the smallest time slot. In our implementation, $interval\_length$ is one month (30 days), and the leaves of this tree represent $slot\_length$ of 5 minutes. To deal with reservations for an arbitrary time T, we first compute a new time which fits into this interval. In order not to overlap reservation from different months, we assume that no reservations are made more than one month in advance. This assumption is also valid for other data structures. As a result, the whole tree can be reused for the next month interval. Hence, the tree is only going to be built once in the beginning.

All operations on Segment Tree are performed recursively. Before giving a brief description of the operations, we define some common notations that will be used, as follows:

- N is the node the recursion is currently in with N_l is the left sibling and N_r is the right sibling.
- $(l_N,\ r_N]$ is the interval of the node N.
- $(l,\ r,\ numCN)$ is the input to all the operations.
- $maxCN$ is the maximum number of available CNs in the system.

For the *search* operation, if a reservation request covers the entire interval of the current node, such that $(l,\ r] == (l_N,\ r_N]$ && $(rv + mv + numCN) \leq maxCN$, then we have found enough free CNs and can terminate the recursion, as shown in Figure 5.2. Hence, Segment Tree is able to search quickly without having to go down to the leaf nodes for a larger interval.

Likewise, for the *add* operation, if $(l,\ r] == (l_N,\ r_N]$, we increase rv by $numCN$ and return $(rv+mv)$ to the parent node. Figure 5.2 shows how the reservations are added into the tree. By using Figure 5.1 as an example, *User1* is stored into node (a), *User2* to node (b) and (d), *User3* to node (c) and (e), and *User4* to node (g). Moreover, the values of rv and mv on each node are updated accordingly. Removing a reservation is very similar to adding one, so the description can be omitted from this chapter.

Algorithm 4: $suggestInterval(l,\ r,\ numCN)$ in Segment Tree

```
1  if numCN > N_availableCN then return −1;
2  if (l, r] == (l_N, r_N] then return l_N;
3  else
4  |  if N is a leaf node then return l;
5  |  if l ∈ (l_{N_l}, r_{N_l}] and r ∈ (l_{N_r}, r_{N_r}] then
6  |  |  leftS ← N_l.suggestInterval(l, l_{N_l}, numCN);
7  |  |  if leftS == l then
8  |  |  |  rightS ← N_r.suggestInterval(l_{N_r}, l_{N_r} + Δ − (l − l_{N_r}), numCN);
9  |  |  |  if rightS == l_{N_r} then return l;
10 |  |  |  else return rightS;
11 |  |  else return N.suggestInterval(leftS, leftS + Δ, numCN);
12 |  else if r ∉ (l_{N_r}, r_{N_r}] then
13 |  |  leftS ← N_l.suggestInterval(l, l_{N_l}, numCN);
14 |  |  if leftS == l then return l;
15 |  |  else return N.suggestInterval(leftS, leftS + Δ, numCN);
16 |  else return N_r.suggestInterval(l, r, numCN);
17 end
```

Searching for a free slot. Brodnik et al. [17] do not describe the operation of finding a new free interval, closest to the proposed reservation $reserv(l,\ r,\ numCN)$, so we give a more detailed description of the implementation of this function. We have to point out that the operation described below finds the closest interval later than the current proposed interval. The description is given in pseudocode in Algorithm 4, and uses the common symbols defined as:

- $N_{availableCN}$ is the number of available CNs in the whole interval of the node N;
- $leftS,\ rightS$ are temporary variables, that store the suggested starting time from the left and right subtree respectively; and
- Δ is the length of the reservation interval, so simply $\Delta = r - l$.

The function recursively searches for a suitable interval. In the case where the reservation interval covers the whole interval of the current node N, it examines the number of available CNs in this interval (lines 1–2). If there are enough CNs, the function returns the leftmost point of the interval l_N, and the rightmost point r_N, otherwise. When the searched interval does not cover the entire interval of the current node (lines 3–17), the function deals with four different possibilities:

1. Current node is a leaf (line 4). This is the boundary condition where the interval is a candidate for the free slot.
2. The interval $(l,\ r]$ covers the intervals of both the node N_l and N_r (lines 5–11). First it finds a candidate interval in the left sibling ($leftS$). If the suggested interval is equal to the original interval (starting at l) we can check if there is enough space in the right subtree as well. Otherwise we re-check the interval $(l_N,\ r_N]$ with a new proposed interval $(leftS,\ leftS + \Delta]$.
3. The interval $(l,\ r]$ covers only the interval of the node N_l (lines 12–15). Similarly to the approach in the first case (above), the procedure searches the left subtree. If the suggested interval is the same as the proposed one, we return it, otherwise we re-check the interval $(l_N,\ r_N]$.

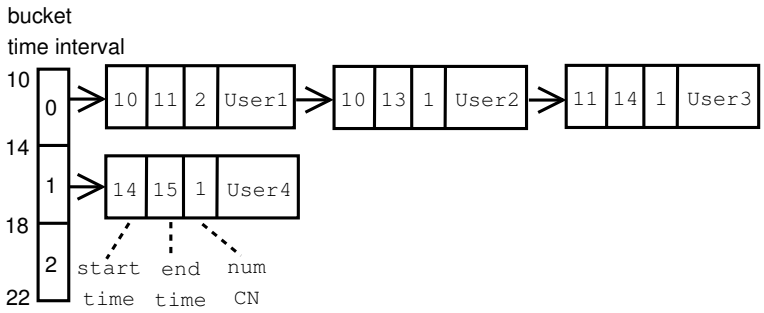


Figure 5.3: A representation of storing reservations in Calendar Queue with $\delta = 4$.

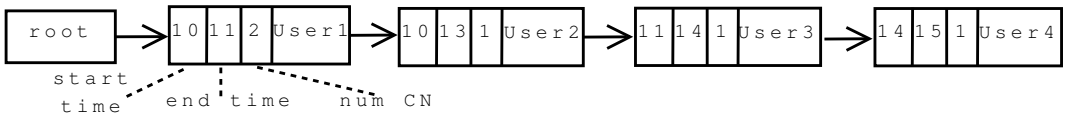


Figure 5.4: A representation of storing reservations in Linked List.

4. The interval $(l,\ r]$ covers only the interval of the node N_r (line 16). In this case we recursively search for a free slot only in the right subtree.

In the case where there is no free interval in Segment Tree, the function returns (-1).

5.2.2 Calendar Queue

Calendar Queue (CalendarQ) was introduced by Brown [18], as a priority queue for future event set problems in discrete event simulation. It is modeled after a desk calendar, where each day or page contains sorted events to be scheduled on that period of time. Hence, CalendarQ is represented as one or more pages or "bucket" with a fixed time interval or width δ. Then, each bucket contains a sorted linked list storing future events. Figure 5.3 shows how reservations are stored in CalendarQ with $\delta = 4$ time interval, by using the example illustrated in Figure 5.1. If a reservation requires more than δ, this reservation will also be duplicated into the next buckets. This approach makes the *search* operation easier since it only searches for a list inside each bucket.

In our implementation, we opted for a static CalendarQ where the number of buckets M and δ are fixed. Hence, these parameters do not need to be adjusted periodically as the queue grows and shrinks. Therefore, by choosing the proper settings for M and δ, CalendarQ performs constant expected time per event processed [45]. In addition, with the static approach, the whole CalendarQ can be reused for the next time period, similar

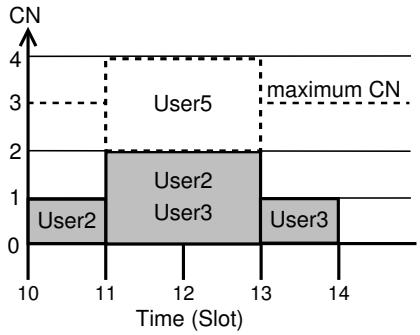


Figure 5.5: A histogram for searching the available CNs in Linked List for *User5*. A dotted box means a new request.

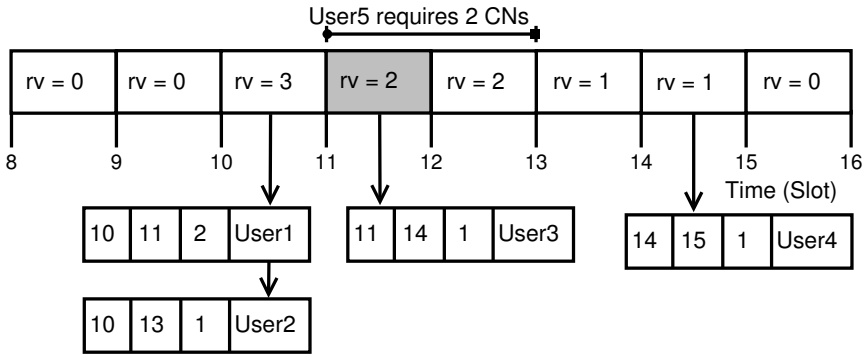


Figure 5.6: A representation of storing reservations in GarQ with Sorted Queue and $\delta = 1$. A request from *User5* is rejected because not enough CNs for slot [11, 12) as shown by the shaded box.

to Segment Tree.

Overall, CalendarQ has a complexity of $O(k)$ for adding reservations, where k is the number of reservations in the list for each bucket. Deleting reservations require a fast $O(1)$ because the reservations are sorted in the list, and CalendarQ only removes the reservations in the current bucket as time progresses. Searching for available CNs require $O(k\ m_{sub})$, where m_{sub} is the number of buckets within a subinterval. The *interval search* operation is the same as the *search* procedure, but it has a larger time interval.

5.2.3 Linked List

Linked List is the simplest and most flexible data structure of all, because accepted reservations will be inserted into the list based on their starting time. In Linked List, each node contains a tuple $\langle t_s,\ t_e,\ numCN,\ user \rangle$. Figure 5.4 shows how these reservations are stored by using the example illustrated in Figure 5.1.

Searching for available CNs. For a *search* operation, the implementation in Linked List is as follows. First, the List needs to find out which nodes have already reserved CNs within $[t_s,\ t_e]$ of the new request. By using the example illustrated in Figure 5.1, we find that only *User2* and *User3* reserve these CNs within the time interval of *User5*. Second, it creates a temporary array for storing the number of CNs used within each time slot, including the new request as shown in Figure 5.5. Finally, it checks each time slot for sufficient number of available CNs. Therefore, for the *search* operation, Linked List has $O(tot_{AR} \quad m_{sub})$, where tot_{AR} is the total number of reservations, and m_{sub} is the number of slots in the subinterval. The same approach also applied to the *interval search* operation, but shifting the time interval to $[t_s + \lambda,\ t_e + \lambda]$ instead, where λ is the length of busy period found from the previous *search* operation. The *interval search* operation ends when it reaches the tail of the List and/or $(t_e + \lambda) > (t_s + MAX\_LIMIT)$, where MAX_LIMIT denotes the maximum time needed for searching.

Adding and Deleting a reservation. These operations can be performed easily in Linked List by iterating through the list from the root node, and comparing each existing node based on its t_s. If the correct position or node has been found, then addition or deletion can be done respectively. Overall, List has $O(tot_{AR})$ complexity for *add* and *delete* operations. However, Linked List can become very inefficient for running these operations on many short reservations, because it needs to find the correct position or node starting from the root node.

5.3 The Proposed Data Structure: Grid Advance Reservation Queue (GarQ)

After analyzing the characteristics of the modified Segment Tree and Calendar Queue data structures in the previous section, we propose an array-based structure for managing reservations in Grid computing. The idea behind this data structure was partially inspired by Calendar Queue and Segment Tree. By combining Calendar Queue and Segment Tree into this structure, we gained the following advantages:

- ability to add new reservations directly into a particular bucket. Hence, it has a fast

$O(1)$ access to the bucket;

- ability to reuse these buckets for the next time period;
- built only once in the beginning;
- easy to search and compare by using iteration;
- easy to implement in comparison to Segment Tree and Calendar Queue; and
- flexibility in handling resource availability. In Grids, CNs can be added or removed periodically. This issue can be addressed by a reservation system or a resource scheduler by setting the amount of available CNs on that resource appropriately. Moreover, existing reservations can be relocated to other CNs through the *add* and *delete* operations.

The proposed data structure has buckets with a fixed δ, which represents the smallest slot duration, as with the Calendar Queue. Each bucket contains rv (the number of already reserved CNs in this bucket) and a linked list (sorted or unsorted), containing the reservations which start in this time bucket. Figure 5.6 shows how reservations are stored in "GarQ with Sorted Queue" with a $\delta = 1$ time interval, by using the example illustrated in Figure 5.1. For enabling a fast $O(1)$ access to a particular bucket, we use the following formula:

$$i = \left\lceil \frac{t}{\delta} \right\rceil \mod M \tag{5.2}$$

where i is the bucket index, t is the request time, and M is the number of buckets in the data structure.

In what follows, we give a detailed description of the four operations: searching for available CNs, adding a reservation, deleting a reservation and searching for the closest free interval. Throughout the description of these operations, a common input for all of them is the tuple $\langle t_s,\ t_e,\ numCN \rangle$. Moreover, they use *start_bucket* and *end_bucket*, which denote the index of the start and end bucket in the reservation interval respectively. To determine the exact index, *get_bucket_index*() function uses Equation 5.2. We also use $maxCN$ to indicate the maximum number of CNs available in the system.

Algorithm 5: $searchReserv(t_s,\ t_e,\ numCN)$ in GarQ

```
1  start_bucket ← get_bucket_index(t_s);                  // get the starting index
2  end_bucket ← get_bucket_index(t_e);                    // get the ending index
3  finish ← 0;
   // a case where it needs to wrap around the array
4  if end_bucket < start_bucket then finish ← M;          // set to the last index
5  else finish ← end_bucket;
6  for i = start_bucket to finish do
   |  // wrapping the array
7  |  if i == M then
8  |  |  i ← 0;                                          // set to the first index
9  |  |  finish ← end_bucket;
10 |  end
11 |  if bucket[i].rv + numCN > maxCN then return false;   // slot is full
12 end
13 return true;
```

5.3.1 Searching for Available Nodes

With GarQ, searching for available CNs is done by iterating through the entire interval and checking each bucket for free CNs, as shown in Algorithm 5. When i points to the end of the array or M, then it needs to search from the beginning of the array (line 7–10). Overall, the complexity of GarQ for searching is $O(m_{sub})$, where m_{sub} is the number of buckets within a subinterval.

Algorithm 6: $addReserv(t_s,\ t_e,\ numCN,\ user)$ in GarQ

```
1  start_bucket ← get_bucket_index(t_s);                  // get the starting index
2  end_bucket ← get_bucket_index(t_e);                    // get the ending index
3  bucket[start_bucket].addInfo(user);                    // store user's jobs & other details
4  finish ← 0;
   // a case where it needs to wrap around the array
5  if end_bucket < start_bucket then finish ← M;          // set to the last index
6  else finish ← end_bucket;
7  for i = start_bucket to finish do
   |  // wrapping the array
8  |  if i == M then
9  |  |  i ← 0;                                          // set to the first index
10 |  |  finish ← end_bucket;
11 |  end
12 |  bucket[i].rv ← bucket[i].rv + numCN;               // increase rv
13 end
```

5.3.2 Adding and Deleting a Reservation

We assume there are enough CNs to add this reservation, i.e. a search has been done beforehand. Adding a new reservation is very similar to searching, and it is described in Algorithm 6. Hence, the complexity of our structure for addition is $O(m_{sub})$ or $O(k+m_{sub})$ when using unsorted and sorted queue respectively, where k is the number of reservations in a bucket list.

Deleting an existing reservation applies to the same principle as adding a new one. It can be done by removing the reservation from the starting bucket and decrementing rv through out the relevant bucket interval.

Algorithm 7: $suggestInterval(t_s,\ t_e,\ numCN)$ in GarQ

```
1  start_bucket ← get_bucket_index(t_s);                    // get the starting index
2  end_bucket ← get_bucket_index(t_e);                      // get the ending index
3  tot_req ← 1 + end_bucket − start_bucket;                 // total slots required
4  new_start ← start_bucket;                                // the new starting index
5  count ← 0;                                  // count number of slots available so far
6  last_bucket ← get_bucket_index(t_s + MAX_LIMIT);   // the last bucket to search
7  finish ← 0;
   // a case where it needs to wrap around the array
8  if last_bucket < start_bucket then finish ← M;           // set to the last index
9  else finish ← last_bucket;
10 for i = start_bucket to finish do
   |  // wrapping the array
11 |  if i == M then
12 |  |  i ← 0;                                            // set to the first index
13 |  |  finish ← last_bucket;
14 |  end
15 |  if bucket[i].rv + numCN > maxCN then
16 |  |  new_start ← i + 1;                          // points to the next bucket
17 |  |  count ← 0;                                  // reset the counter to zero
18 |  else count ← count + 1;
19 |  if count >= tot_req then break;          // exit loop if found enough slots
20 end
21 if count < tot_req then new_start ← (−1);  // all slots do not have enough CNs
22 new_time ← convert_index(new_start);    // convert bucket index into start time
23 return new_time;
```

5.3.3 Searching for a Free Time Slot

Searching for the closest interval is also straightforward in GarQ, as shown in Algorithm 7. This algorithm is similar to Algorithm 5, but the search interval is now expanded by

Table 5.1: Summary of the data structures, where n is the number of tree nodes, k is the number of reservations in the list for each bucket, m_{sub} is the number of buckets or slots within a subinterval, and tot_{AR} is the total number of reservations.

Name	Time Complexity		
	Add	Delete	Search
Segment Tree	$O(log\ n)$	$O(log\ n)$	$O(log\ n)$
Calendar Queue	$O(k)$	$O(1)$	$O(k\ m_{sub})$
Linked List	$O(tot_{AR})$	$O(tot_{AR})$	$O(tot_{AR}\ \ m_{sub})$
GarQ with Unsorted Queue	$O(m_{sub})$	$O(k + m_{sub})$	$O(m_{sub})$
GarQ with Sorted Queue	$O(k + m_{sub})$	$O(m_{sub})$	$O(m_{sub})$

MAX_LIMIT. This constant variable denotes the maximum time needed for the *interval search* operation, hence, it prevents the algorithm from searching the array infinitely. During the searching, a temporary counter *count* indicates how many buckets still need to be searched (and have enough free CNs) before the operation can finish (line 15–19). At the end of the operation, the index of a new start bucket, *new_start*, is converted into the new starting time by using *convert_index*() function.

After describing these data structures, a summary of each of them is given in Table 5.1, including our proposed data structure, namely GarQ with either Unsorted or Sorted Queue. In the next section, we evaluate the performance of our data structure with the existing ones. We conduct the evaluation using real workload traces taken from production systems.

5.4 Performance Evaluation

In order to evaluate the performance of our proposed data structure, i.e. GarQ with Unsorted Queue (GarQ-U) and GarQ with Sorted Queue (GarQ-S), we compare them to Linked List (List), Segment Tree (Tree) with $slot\_length = 5$ minutes, and static Calendar Queue (SCQ) with $\delta = 1$ hour. For SCQ to be optimal, we choose the value of δ based on the jobs' average duration time as stated in Table 5.2. For GarQ-U and GarQ-S, we set each slot to be a 5-minute interval. All, except List, have a fixed interval length of 30 days, as mentioned previously. Finally, we simulate a homogeneous cluster of 64 compute nodes, i.e. $maxCN = 64$.

For the evaluation, we are investigating: *(i)* the total number of tree nodes or slots

Table 5.2: Workload traces used in this experiment.

Trace Name	Location	# Jobs	Mean Job Time	From	To
DAS2 fs0	Vrije Univ., The Netherlands	225,711	11.74 minutes	Jan 2003	Dec 2003
LPC EGEE	Clermont-Ferrand, France	242,695	52.07 minutes	Aug 2004	May 2005
SDSC BLUE	San Diego, USA	243,314	69.34 minutes	Apr 2000	Jan 2003

accessed throughout for each of the operations, including temporary ones for List and SCQ; *(ii)* the average runtime for using the above operations; and *(iii)* the average memory consumption for these data structures. Note that we conduct the experiment this way because we want to get a clear picture on how each data structure performs, without the interference of external factors or scheduling issues, such as deadline, backfilling and job preemption.

5.4.1 Experimental Setup

We selected three workload traces from the Parallel Workload Archive [49] for our experiments, as summarized in Table 5.2. These traces were chosen because they represent a large number of jobs and contain a mixture of single and parallel jobs. In addition, the LPC trace was based on recorded activities from the EGEE (Enabling Grids for E-science in Europe) project, hence it is very suitable for conducting the evaluation. Moreover, as shown in Table 5.2, the average job duration time varies from 11 to 70 minutes. Hence, we can analyze in more detail the performance of each data structure for jobs with a short, medium and long duration time.

Although these traces were taken from the real production systems, the jobs' start time were logged in increasing order. Thus, it might not be suitable for testing out the *interval search* operation. Therefore, we *shuffled* or *randomized* the start time order of jobs for every 2-week period of each trace. Overall, we have 6 traces in this experiment: the 3 original ones and 3 shuffled ones. Several modifications have also been made to these traces, as follows:

- If a job requires more than the total number of nodes of a resource, we set this job to $maxCN$.
- A request's start time is rounded up to the nearest 5-minute time interval. For example, if a job request starts at time 01:03:05 (hh:mm:ss), then it will be rounded

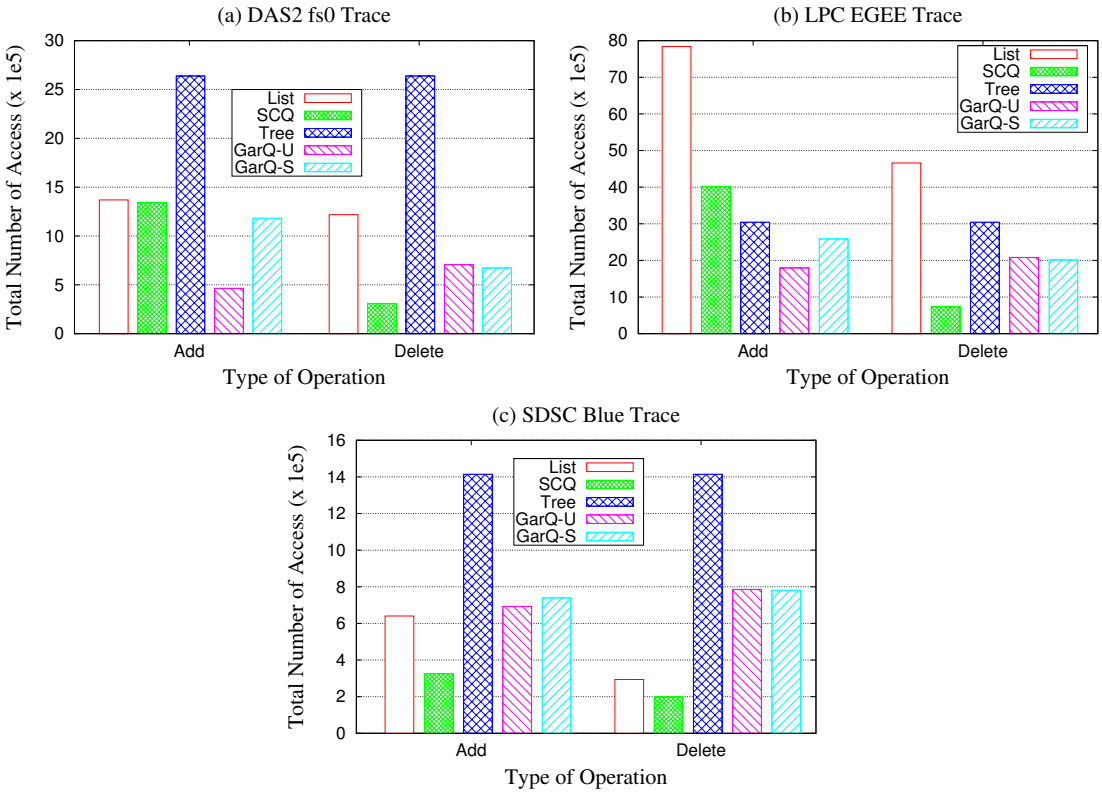


Figure 5.7: Total number of nodes accessed during *add* and *delete* operation using original traces (lower number is better).

to time 01:05:00.

- A job duration time is within the range of 5 minutes to 28 days. We limit the maximum duration time to prevent overlapping reservations from different months. Hence, each structure, except for Linked List, can be reused and built only once.

5.4.2 Experimental Results

Adding and Deleting Reservations

Figures 5.7 and 5.8 show the total number of access for adding and deleting reservations using the original and the shuffled traces, respectively. Note that List has been omitted in Figure 5.8 due to a much greater number of access than other structures, by at least 60-fold from SCQ.

For the *add* operation, GarQ-U performs the best compared to other structures, as shown in Figure 5.7 (a) and (b). The main reason is that when there are many reservations stored in a slot, GarQ-U does not need to sort them. Thus, GarQ-U reduces the number

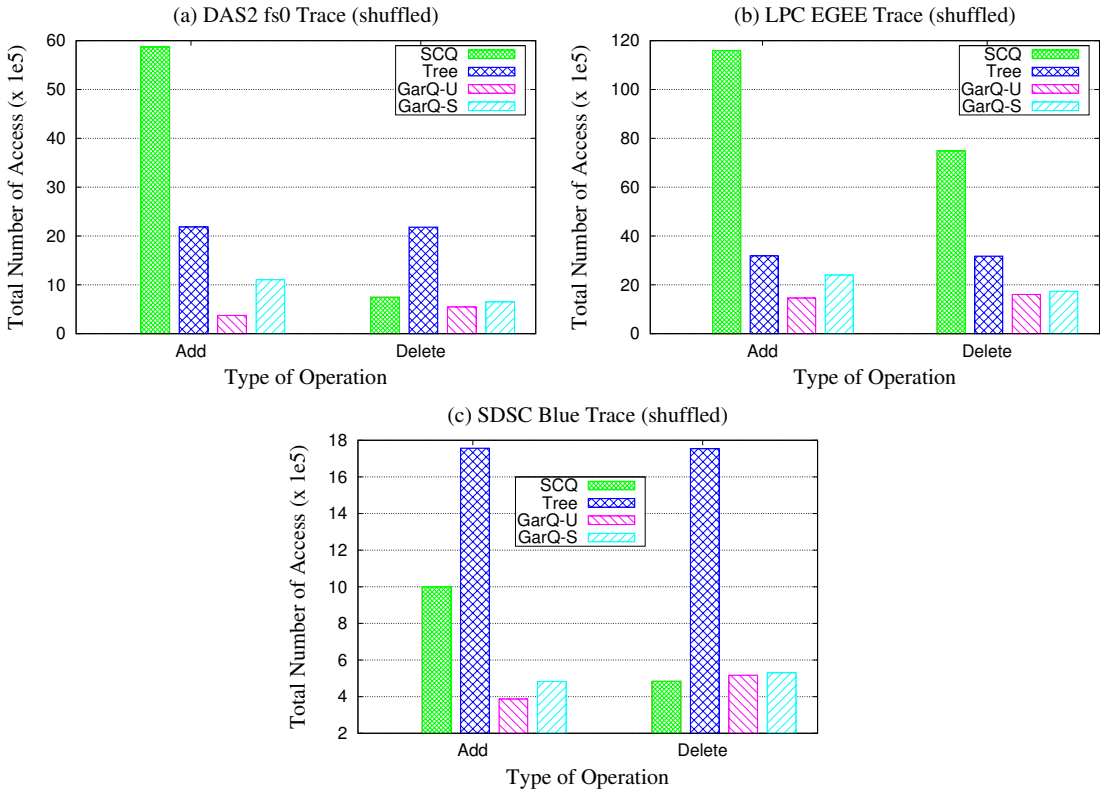


Figure 5.8: Total number of nodes accessed during *add* and *delete* operation using shuffled traces (lower number is better).

of access by at least 150% and 44% compared to GarQ-S for the DAS2 and LPC traces, respectively. GarQ-U achieves a much lower number of access than GarQ-S for the DAS2 trace compared to the LPC trace, since the DAS2 trace contains many small jobs. As a result, GarQ-U avoids the overhead of sorting many reservations in a particular bucket.

A similar trend is also noted for the *add* operation using the shuffled traces of DAS2 and LPC, as shown in Figure 5.8 (a) and (b), respectively. GarQ-U manages to reduce the number of access by at least 194% and 64% compared to GarQ-S for the DAS2 and LPC shuffled traces, respectively.

For large jobs in the SDSC trace, GarQ-U has a similar performance to GarQ-S, as shown in Figure 5.7 (c). However, SCQ is able to reduce the number of access by more than a half compared to GarQ-U and GarQ-S. List also performs better than GarQ-U and GarQ-S by at least 8%. The main reason is because both SCQ and List append new requests at the end since these requests arrive sequentially. If they arrive randomly, as shown in Figure 5.8 (c), GarQ-U and GarQ-S are able to lower the number of access by

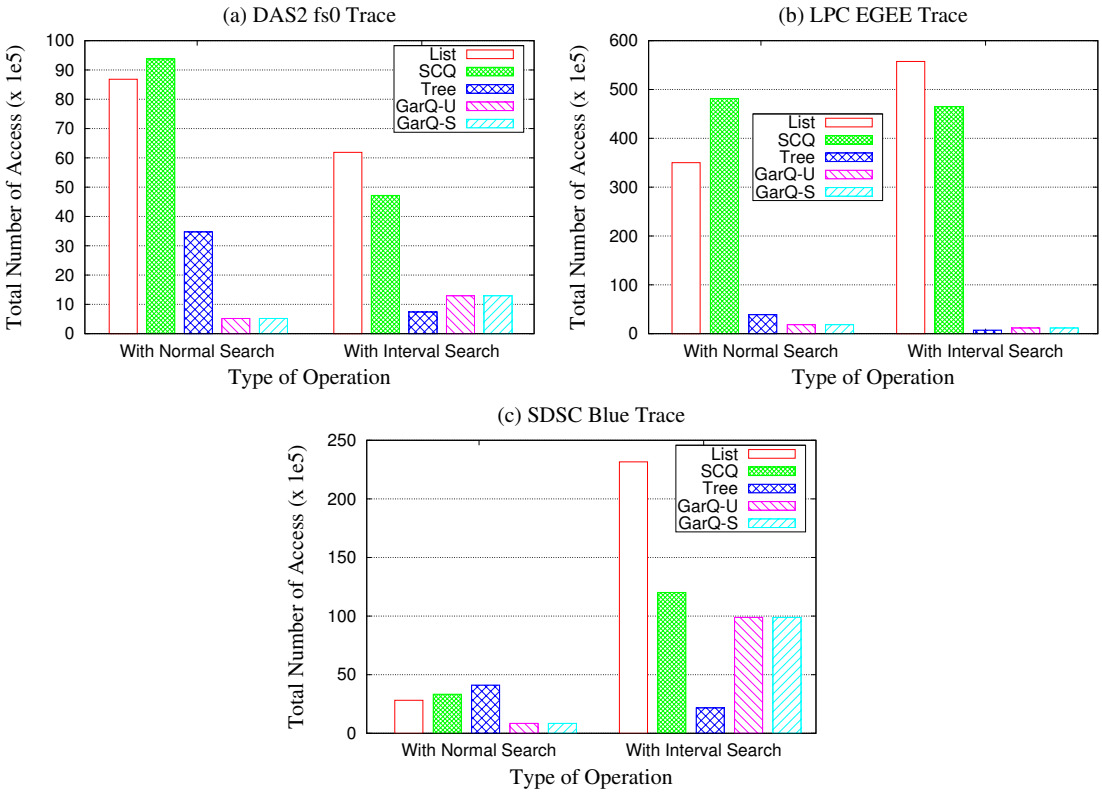


Figure 5.9: Total number of nodes accessed during *search* operations using original traces (lower number is better).

more than a half compared to SCQ. In fact, for all the shuffled traces, both GarQ-U and GarQ-S are always better than Tree, SCQ and List.

Theoretically, when it comes to deleting existing reservations, SCQ with the $O(1)$ time complexity should have the best performance. This is because SCQ only deletes reservations in the particular array bucket. Thus, Figure 5.7 clearly shows the superiority of SCQ compared to other structures for the *delete* operation. More specifically, SCQ is able to reduce the number of access by more than a half compared to GarQ-U and GarQ-S.

In Figure 5.7 (c), List also performs better than GarQ-U and GarQ-S by more than a half. This is mainly due to deleting reservations that are located at the front of the list, since they are arriving sequentially. In addition, since the SDSC trace contains many large jobs, both GarQ-U and GarQ-S need to decrement rv on buckets located within the given time interval, thus, giving additional number of access.

On the other hand, for the shuffled traces in Figure 5.8 (a) and (c), the performances of GarQ-U and GarQ-S for the *delete* operation are shown to be on par with SCQ. For

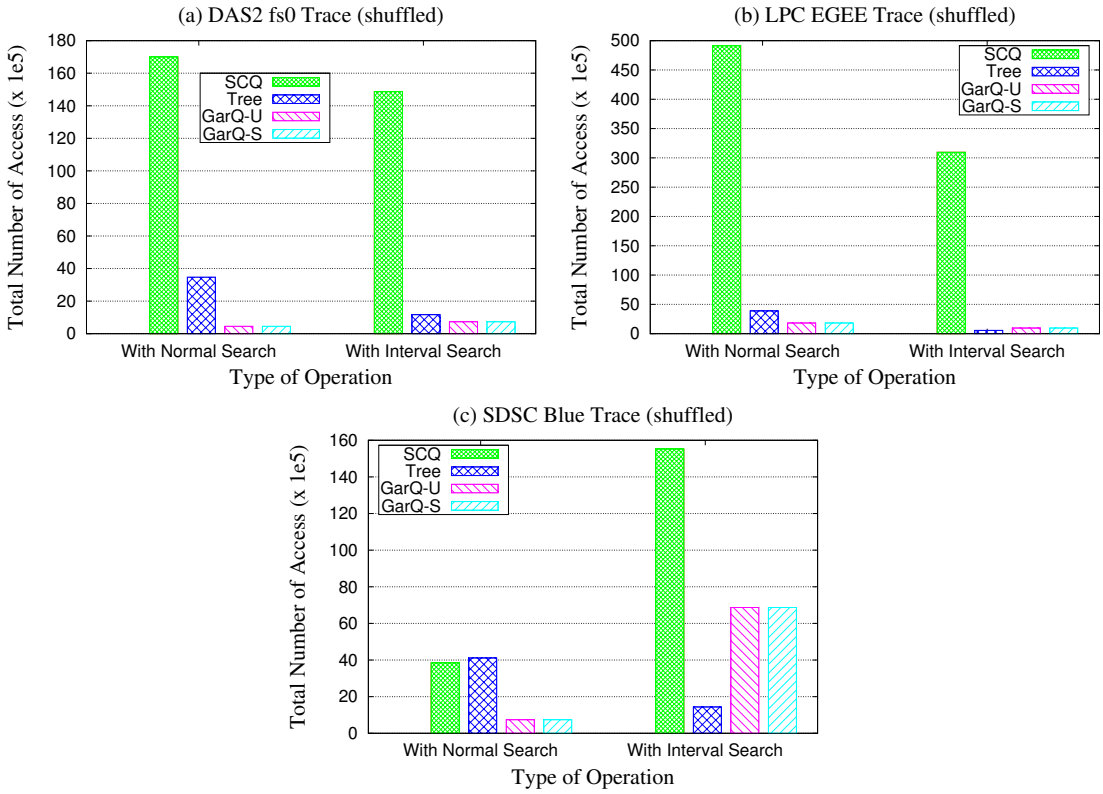


Figure 5.10: Total number of nodes accessed during *search* operations using shuffled traces (lower number is better).

the shuffled LPC trace, as depicted in Figure 5.8 (b), SCQ performs worse because in each bucket, the incoming reservations are sorted based on their start time. In the worst case scenario, some reservations located in front of the list have a longer duration. Thus, SCQ needs to iterate through the list to remove completed reservations that have a shorter duration time.

Searching for Available Slots

Figures 5.9 and 5.10 show the total number of nodes accessed when searching for empty slots using the original and shuffled traces respectively. Note that for the *interval search* operation, we set the maximum time limit or MAX_LIMIT to be 12 hours from the request's initial start time. In addition, the results for List has been omitted in Figure 5.10, since it has a much greater number of access than other structures.

For the *normal search* operation, Figures 5.9 and 5.10 show that GarQ-U and GarQ-S have the best performance. This is because they perform a sequential and straightforward

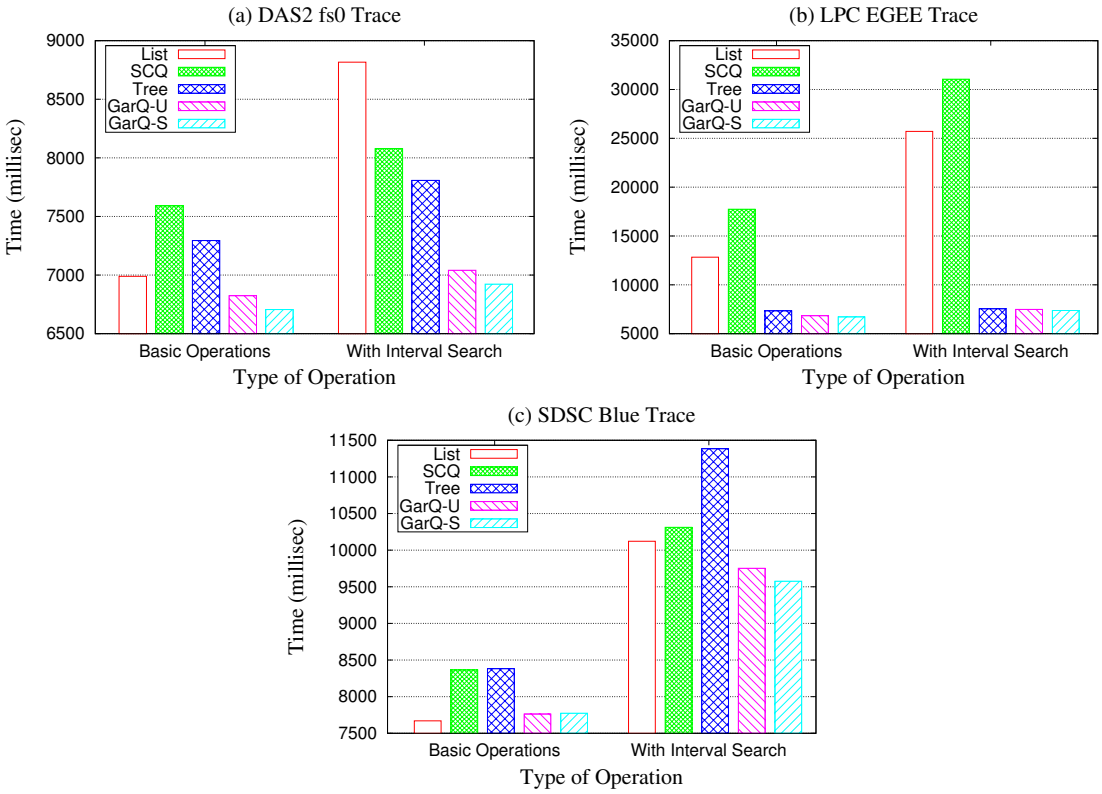


Figure 5.11: Average runtime using original traces (shorter time is better).

comparison. Thus, they have the same number of access for this operation. In contrast, Tree has to traverse down to the left and/or right subtrees, and thus, visits many nodes along the search path. In the worst case scenario, Tree needs to traverse down to the leaf nodes to search for available resources for small jobs. List and SCQ perform the worst as they have to start searching from the beginning of a list, and iterate through the affected reservations.

For the *interval search* operation, Tree has an advantage over GarQ-U and GarQ-S, since it can find out the resource availability at a larger time interval and with fewer number of nodes to visit. This scenario is clearly shown for the SDSC trace, as depicted in Figures 5.9 (c) and 5.10 (c).

Average Runtime Performance

To measure the average runtime performance of each data structure, we run the experiments several times on a 2 Ghz Opteron machine with 4 GB of RAM. We take into account the time required to perform "basic operations", i.e. conducting the *add, delete*

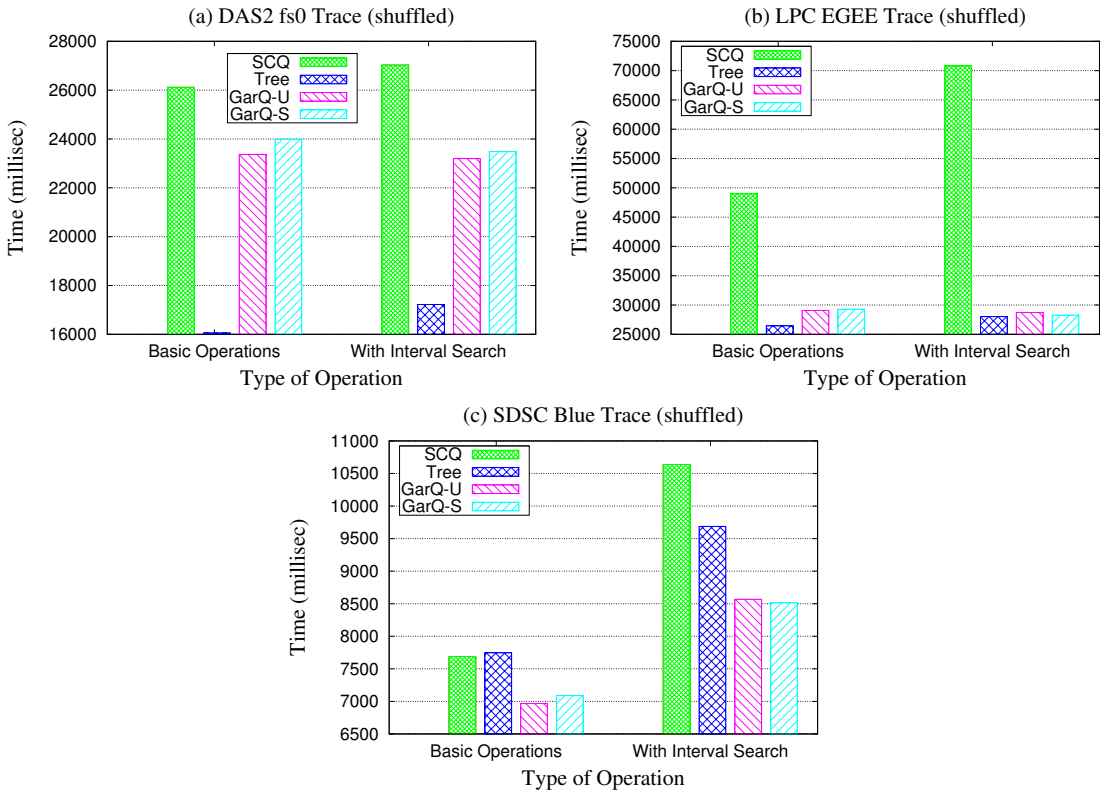


Figure 5.12: Average runtime using shuffled traces (shorter time is better).

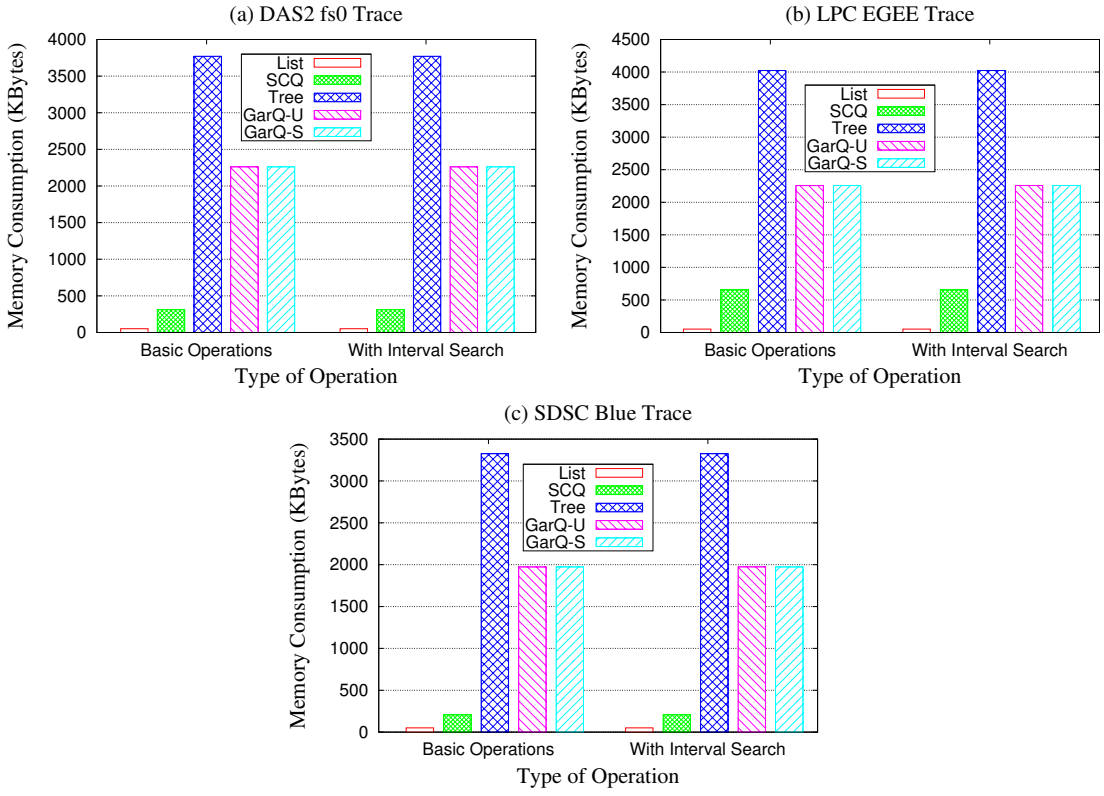


Figure 5.13: Average memory consumption using original traces (lower memory is better).

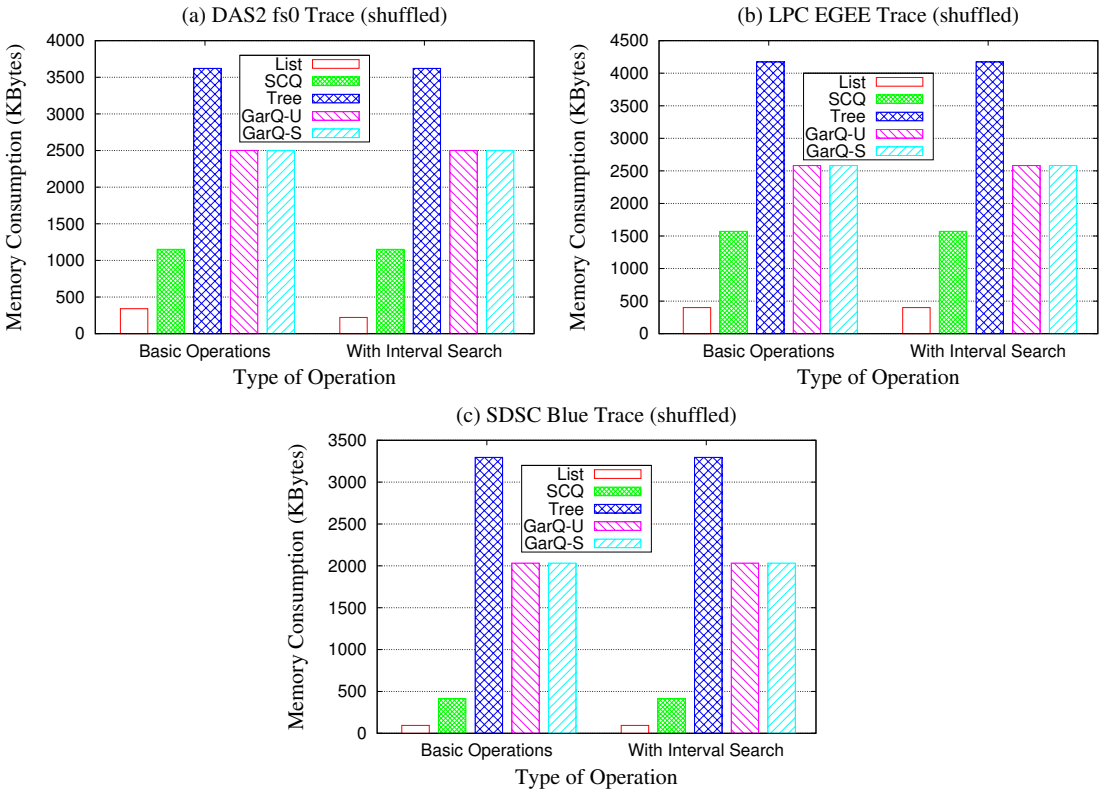


Figure 5.14: Average memory consumption using shuffled traces (lower memory is better).

and *search* operation as a whole, and to run these operations using only the *interval search*. Figures 5.11 and 5.12 show the average runtime using the original and the shuffled traces, respectively. Note that the results for List has been omitted in Figure 5.12, since it has a much greater number of access than other structures.

For the basic operations, as shown in Figure 5.11, GarQ-U and GarQ-S perform the best overall, whereas SCQ performs the worst. For SCQ, the δ value of 60 minutes is not optimal to manage small and medium jobs of DAS2 and LPC traces respectively, as shown in Figure 5.11 (a) and (b). This is because most of these jobs are concentrated in a particular bucket, and not spread out to other buckets.

For operations that include only the *interval search*, as shown in Figure 5.11, GarQ-U and GarQ-S perform the best overall. For Figure 5.11 (a) and (b), List and SCQ perform worse than Tree as expected. However, Tree takes the longest time for running large jobs of the SDSC trace, as shown in Figure 5.11 (c). This is partly due to the overhead of using recursive functions.

In Figure 5.12 (a), GarQ-U and GarQ-S do not perform too well compared to Tree

because this trace contains many small jobs. SCQ also takes a big performance hit for managing these jobs. An improvement to GarQ can be done by imposing a minimum duration limit by the resource and/or grouping small jobs as one big batch before requesting a reservation. With this approach, GarQ will be able to perform more efficiently, since this scenario will be similar to reserving large jobs, as shown in Figure 5.12 (c). It is also important to note, on average, the overhead cost of using the interval search operation in GarQ-U and GarQ-S is minimal compared to other structures. This is a very encouraging result since the array-based implementation is also easy to implement.

Average Memory Consumption

For measuring the average memory consumption of each data structure, we run the experiments on the same setup as previously mentioned, i.e. using the 2 Ghz Opteron machine with 4 GB of RAM. We measure the memory consumption based on the measurement before and after the experiment. Moreover, in order to improve accuracy, we run the experiment several times.

From Figures 5.13 and 5.14, List and SCQ are very efficient in all of the traces, followed by GarQ-U and GarQ-S. However, SCQ requires more memory than List due to the cost of having fixed M buckets, and duplicating reservations that take longer than δ across several buckets. Tree consumes more memory because the complete structure needs to be built for the entire length of time interval we want to cover. Note that in these experiments, all data structures require less than 5 KB of RAM in a machine with a total RAM of 4 GB. Therefore, the trade-off between space and time complexity can be neglected.

On the other hand, there is a big trade-off between low memory consumption and runtime performance. Even though both List and SCQ consume the least amount of memory, their runtime performance were the worst, as mentioned previously. In contrast, Tree consumes more memory, but runs faster than List and SCQ. Finally, GarQ-U and GarQ-S have a moderate memory consumption, but a better runtime performance compared to List, Tree and SCQ (on average). Overall, GarQ-U and GarQ-S have a better ratio.

In terms of comparing GarQ-U with GarQ-S, both have a similar ratio and an equal number of access in the *search* operations. However, GarQ-U performs better than GarQ-S in the *add* operation. Thus, for the remaining of this thesis, we refer GarQ with Unsorted

Queue (GarQ-U) as GarQ.

5.5 Summary

An efficient data structure is important for minimizing the time complexity needed to perform advance reservation operations, such as searching for available resources, adding new requests and deleting existing reservations. This chapter proposes a new data structure, named Grid advance reservation Queue (GarQ), for administering reservations efficiently. In addition, this chapter introduces a new operation, called *interval search*, to find a free time interval closest to the requested reservation, if it was previously rejected. This operation has a significant value to users, because it locates the next suitable reservation time.

GarQ is an array-based data structure inspired by Calendar Queue and Segment Tree. According to our performance evaluation, whose input is taken from real workload traces, such as DAS2 fs0 from Vrije University in the Netherlands, GarQ manages to perform much better on average than Linked List, Segment Tree and Calendar Queue for the above reservation operations. However, for small jobs in the randomized DAS2 fs0 trace, Segment Tree proves to have the best average runtime performance. We shuffled or randomized the starting time of jobs from these traces because they are logged in increasing order of arrival time. Overall, GarQ has a better ratio between low memory consumption and runtime performance compared to these data structures. Hence, the results of GarQ are encouraging because it is also easy to implement and can be reused for the next time interval. Therefore, GarQ only needs to be built once in the beginning. In the next chapter, we present an elastic reservation model for Grid systems, and show how GarQ is used by an on-line strip packing algorithm to find alternative reservation offers.

Chapter 6

Elastic Reservation Model with On-line Strip Packing Algorithm

This thesis provides a case for an elastic reservation model, where users can *self-select* or choose the best option in reserving their jobs, according to their Quality of Service (QoS) needs, such as deadline and budget. In addition, this thesis adapts an on-line strip packing algorithm to provide alternative offers, and reduce fragmentations or idle time gaps caused by having reservations in the system.

6.1 Introduction

In order to reserve the available resources, a user must first submit a request by specifying a series of parameters such as number of resources needed, and start time and duration of his/her jobs [86]. Then, the system checks for the feasibility of this request. If one or more parameters can not be satisfied, then the request is rejected. Hence, this approach is known as an *inelastic* or *rigid* method, because these parameters are hard constraints that do not permit the system any modifications.

Consequently, the user may resubmit new requests with modified existing parameters, such as a different start time and/or duration until available resources can be found. However, this approach will have a negative impact in increasing the communication overheads between users and the resource. Moreover, it will also degrade the performance of the re-

source in managing many incoming requests due to previously rejected ones. Finally, if such a solution is found, it might not be a good one since it only looks for the first available resources. As a result, it will cause *fragmentations* of AR jobs, which leave behind many gaps of idle time among them. Thus, the resource utilization will be significantly lowered.

To overcome the above problem, this thesis introduces an *elastic* reservation model, which takes into consideration the resource utilization when processing reservation requests. With this model, users can query about the resource availability on a given time interval. They can also provide a reservation duration time and/or number of compute nodes (CNs) needed as soft constraints to the query. Then, the resource will give the users an offer and/or a list of alternative ones, if these constraints can not be met. This approach allows a flexibility to the users to *self-select* or choose the best option in reserving their jobs according to their Quality of Service (QoS) needs, such as deadline and budget. For this model, this thesis adapts an existing on-line strip packing algorithm [40, 90] to provide these alternative offers, and reduce fragmentations or idle time gaps caused by having reservations in the system.

The importance of the self-select or self-service concept is further highlighted by a survey done in 2007 by the International Air Transport Association (IATA) for the airlines industry. The survey was conducted on over 10,000 active travelers. The result shows that 54% of the survey participants said yes to more self-service options, and 69% of them had used the provided self-service kiosks [72]. The result also shows that 83% of these participants wished to have the opportunity to choose their own seats through online websites [72]. In Chapter 7, we consider compute nodes as perishable, similar to aircraft seats. Thus, the IATA findings are notably related and important to our problem domain.

6.2 Description of the Elastic Reservation Model

6.2.1 User Model

In order to reserve compute nodes, a user needs to submit a reservation request. In this model, the request is defined as $reserv(t_s,\ t_e,\ numCN)$, where t_s denotes the reservation start time, t_e denotes the reservation end time, and $numCN$ indicates the number of compute nodes to be reserved, respectively. When the system receives the request, it

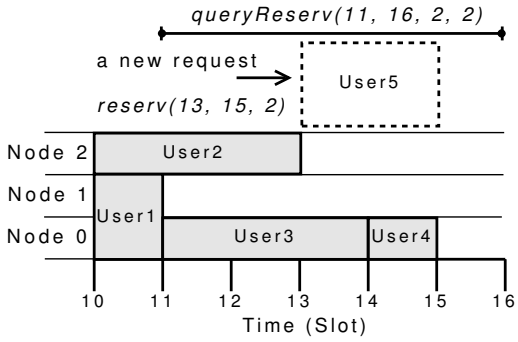


Figure 6.1: An example of elastic AR with 3 nodes. A dotted box denotes a new request.

checks for availability. Then, the system replies back to the user whether it can accept the request or not. If the request has been accepted, then the user sends his/her jobs or goes back to submit a new reservation request with a different time interval.

To increase the chance of getting accepted, the user can query about available time slots. This query operation is defined as $queryReserv(ti_s,\ ti_e,\ dur?,\ numCN?)$, where ti_s denotes the earliest start time interval, ti_e denotes the latest end time interval, and dur denotes the reservation duration time, respectively. Note that the "?" sign indicates that this attribute is optional. In addition, we assume that $(ti_e - ti_s) \geq dur$.

Upon receiving the $queryReserv()$ operation, the resource will find a solution or an offer that satisfies both dur and $numCN$ constraints. Otherwise, these parameters are treated as soft constraints, and a list of alternative offers are given. The list is defined as $offerList[\ ] = \{\ offer(t_s,\ t_e,\ numCN) +\ \}$, where the "+" sign denotes one or more occurrences of this tuple. These offers are temporary results generated from this $queryReserv()$ operation. Thus, the user needs to select an offer and to send a $reserv(t_s,\ t_e,\ numCN)$ operation for a guarantee. Note that in this thesis, we solely focus on reserving homogeneous nodes as the type of resource. Moreover, the "?" and "+" signs are borrowed from a W3C recommendation on XML [16].

Figure 6.1 shows an example of existing reservations in the system, represented as a time-space diagram. When a new query from *User5* arrives, i.e. $queryReserv(11,\ 16,\ 2,\ 2)$, the system checks for any available nodes within $[11,\ 16]$ time interval. It founds a solution, which is $offer(13,\ 15,\ 2)$, that satisfies both dur and $numCN$ constraints. Then, the user sends a reservation request, i.e. $reserv(13,\ 15,\ 2)$, to accept this offer.

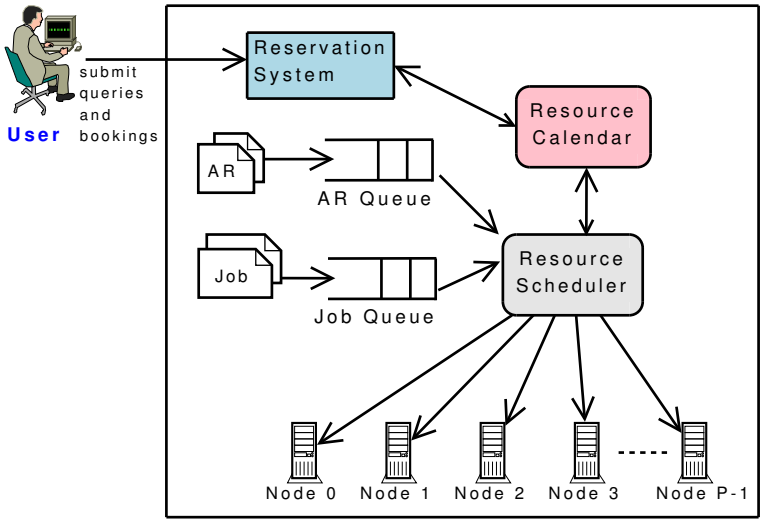


Figure 6.2: System that supports an elastic reservation model.

6.2.2 System Model

To incorporate an elastic reservation model into an existing system (discussed in Chapter 4.3.2), we add two new components: Reservation System and Resource Calendar, as shown in Figure 6.2. The Reservation System handles users' queries and bookings, whereas the Resource Calendar stores reservations' details and updates node availability as time progresses or as needed.

The Reservation System communicates with the Resource Calendar to search for available nodes, and add new reservations. The Resource Scheduler also interacts with the Resource Calendar to determine the start time of reserved jobs in the AR Queue. For the Reservation System, we adapt an on-line strip packing algorithm, which will be discussed next. For the Resource Calendar, we use GarQ, as explained in Chapter 5.

6.3 On-line Strip Packing Algorithm

In this section, we describe how to generate suitable offers for AR requests, by using an adapted on-line strip packing (OSP) algorithm for our elastic model. Since no prior knowledge of AR arrivals is given, the proposed OSP algorithm focuses on finding a solution or alternative offers for each request. Hence, OSP aims to increase resource utilization and reduce fragmentations. This problem is similar to a strip packing problem, where objects of different sizes are coming in and it is trying to minimize the height of objects packed

Algorithm 8: The OSP algorithm for an elastic reservation model.

Input: $queryReserv(ti_s,\ ti_e,\ dur,\ numCN)$
Output: $offerList[\ ]$ or a list of offers, including a solution (if found)

```
1  if (dur ≠ φ) and (numCN ≠ φ) then
2  |  needSol ← true;
3  else needSol ← false;
   // initialize with a default value
4  if (dur == φ) then dur ← δ;
5  if (numCN == φ) then numCN ← 1;
6  offerList[ ] ← φ;
7  ts ← get_total_slot(dur);                                  // total slots needed
8  slotList[ ] ← find_consecutive_slot(ti_s, ti_e);
9  size ← get_size(slotList);                                 // size of list
   // rank slotList[ ] in the increasing order of freeCN and returns its indices
10 indexRank[ ] ← get_sorted_index( slotList[ ] );
   // a loop to search for offers
11 for (i = 0) to (size − 1) do
12 |  index ← indexRank[i];                                   // current index
13 |  slot ← slotList[index];                                 // current slot
   |  // skips this unsuitable slot and goes back to the top of the loop
14 |  if (slot.freeCN < numCN) then continue;
15 |  head ← indexRank[i];                                   // starting index
16 |  tail ← indexRank[i];                                   // ending  index
17 |  totSlot ← slot.numSlot;                  // total number of slots found so far
18 |  minCN ← slot.freeCN;        // lowest freeCN that can be offered so far
   |  // look for slots located earlier than slotList[index] (left side)
19 |  for (l = index − 1) to (l ≥ 0) do // decrement
20 |  |  freeCN ← slotList[l].freeCN;
21 |  |  if (freeCN < numCN) or (totSlot ≥ ts) then
22 |  |  |  break;
23 |  |  end
24 |  |  head ← l;                                  // starts from this slot
25 |  |  totSlot ← totSlot + slotList[l].numSlot;
26 |  |  minCN ← min(freeCN, minCN);
27 |  end
   |  // look for slots located later than slotList[index] (right side)
28 |  for (r = index + 1) to (r ≤ size − 1) do
29 |  |  freeCN ← slotList[r].freeCN;
30 |  |  if (freeCN < numCN) or (totSlot ≥ ts) then
31 |  |  |  break;
32 |  |  end
33 |  |  tail ← r;                                   // ends until this slot
34 |  |  totSlot ← totSlot + slotList[r].numSlot;
35 |  |  minCN ← min(freeCN, minCN);
36 |  end
37 |  offer ← make_offer(head, tail, totSlot, minCN);          // make a new offer
38 |  offerList[ ] ← add_offer(offerList[ ], offer);       // storing list of offers
   |  // found a solution
39 |  if (totSlot ≥ ts) and (needSol == true) then
40 |  |  offerList[ ] ← found_sol(offer, offerList[ ]);
41 |  |  needSol ← false;
42 |  |  break;                  // stops looking for more offers (exit the loop)
43 |  end
44 end
45 offerList[ ] ← set_cost(offerList[ ]);
46 return offerList[ ];
```

into one bin. Applying this problem to our model, an AR request represents an object of which its width and height are $numCN$ and dur respectively.

Algorithm 8 shows the proposed OSP algorithm for each AR request. If the request does not specify any duration time, then the OSP algorithm sets the dur parameter to be δ by default (line 4), where δ is a fixed time interval used by GarQ (as mentioned in Chapter 5). Similarly, $numCN = 1$ if the value of $numCN$ is not given (line 5). If both parameters are specified in the request, the OSP algorithm aims to find a solution that satisfies these constraints. The boolean variable $needSol$ is used to notify such a case (line 1–3), such that in the end, this solution can be placed at the top of the list. If no solution is found, the OSP algorithm treats the dur and $numCN$ parameters as soft constraints.

After getting these constraints, OSP obtains a list of consecutive slots ($slotList[\,]$) from GarQ within the $[ti_s,\ ti_e]$ interval (line 8). We define a *consecutive* slot to be a sequence of slots with the same number of $freeCN$, i.e. $maxCN - slot.rv$, where $maxCN$ is the maximum number of nodes. The aim is to reduce the total number of slots needed to search for available nodes. Then, OSP ranks these consecutive slots in an increasing order of $freeCN$, and stores them in $indexRank[\,]$ (line 10). Therefore, $indexRank[i]$ indicates the index of a slot with the $(i\ +\ 1)$-th low $freeCN$ in $slotList[\,]$ (line 12–13), such that

$$slotList[iA].freeCN\ \leq\ slotList[iB].freeCN$$

where $iA = indexRank[i],\ \ iB = indexRank[i+1]$, and $i = 0,\ \ \ldots,\ size - 1$.

After sorting $slotList[\,]$, OSP iterates $indexRank[\,]$ searching for *a local minima* (line 11–44). We denote a local minima as the first consecutive slot, where its $freeCN$ is greater than or equal to $numCN$. Otherwise, the slot will be ignored by OSP (line 14). The aim of this exercise is to use this consecutive slot first. In the best case scenario, nodes of this slot are fully utilized. As a consequence, other nearby slots can be allocated to run reserved and/or non-reserved jobs that may require more than one node. Thus, OSP takes into consideration the need of users running their parallel jobs in the system.

Then, OSP aims to satisfy the dur (duration time) or ts (total slots needed) constraint. At each consecutive slot, OSP sets $head$ and $tail$ variables to the current position of $indexRank[\,]$ (line 15–16) for later usage, where $head$ denotes the starting index, and $tail$

denotes the ending index. OSP also adds *numSlot* to *totSlot* (line 17), where *numSlot* denotes the number of slots grouped together, and *totSlot* denotes the total number of slots found so far. Finally, OSP sets $minCN$ to the slot's $freeCN$ (line 18) for later usage, where $minCN$ denotes the lowest number of available nodes that can be offered so far.

If *totSlot* is less than ts, then OSP looks for more *numSlot* (line 19–36). First, OSP looks to the left side or finds more slots that are located earlier than $slotList[index]$ (line 19–27), where *index* denotes the position of the consecutive slot in $slotList[\,]$. If *totSlot* is still not enough, OSP looks to the right side or finds more slots that are located later than $slotList[index]$ (line 28–36). For either side, the *head* (for the left side), *tail* (for the right side), *totSlot*, and $minCN$ variables are updated accordingly. The search on either side ends if it satisfies one of the following conditions: (i) the number of available nodes at each slot is less than $numCN$; (ii) the total number of slots found so far equals or exceeds ts; or (iii) the search hits one of the sentinels (beginning or ending position in $slotList[\,]$). The main reason to search the left side first is to have a solution that is closest to the starting time interval (ti_s) given by the user.

After the search ends, OSP makes a new reservation offer (line 37), where *totSlot* is converted into the actual duration time. This offer is within the [*head*, *tail*] interval in $slotList[\,]$, and has $minCN$ available nodes. Subsequently, this offer is added to $offerList$ (line 38), where $offerList$ denotes a list containing newly-created offers. If the total number of slots, *totSlot*, from this offer meets the ts and *needSol* objectives, then this offer is marked as a solution or the most preferred one (line 39–43). Then, the $found\_sol()$ function moves this offer to the top of the list to become the first choice (line 40). In addition, OSP stops looking for more offers if such solution is found.

Once all offers have been made, OSP applies the total cost or price to each of them (line 45). Finally, OSP gives the list to the user (line 46), so he/she can decide. In addition, the user is given the flexibility to reduce the *dur* and/or $numCN$ values of an offer. Overall, the time complexity for this OSP algorithm is $O(n^2)$, where n denotes the number of consecutive slots in $slotList[\,]$. Note that detailed explanations on calculating the price of each offer will be discussed in Chapter 7.

6.4 Performance Evaluation

In order to evaluate the performance of our proposed algorithm, i.e. the On-line Strip Packing (OSP) algorithm, we compare it to a First Fit (FF) algorithm. Moreover, we introduce a Rigid algorithm as a base comparison. The FF algorithm only looks for the first available nodes within a given time interval, whereas the Rigid algorithm treats ti_s, dur and $numCN$ as hard constraints. Therefore, if no solution is found, then the Rigid algorithm will reject such reservation requests. Note that only the OSP algorithm provides a list of offers for this experiment.

For scheduling reserved and non-reserved jobs from the queues, we incorporate First Come First Serve (FCFS) and Easy Backfilling (BF) [98] policies into the Resource Scheduler. Thus, for this experiment, we model a system that uses one of the following Reservation System and Resource Scheduler combinations: FF with FCFS ($FF + FCFS$), FF with BF ($FF + BF$), OSP with BF ($OSP + BF$), Rigid with FCFS ($Rigid + FCFS$), and Rigid with BF ($Rigid + BF$). In addition, the system uses GarQ for the Resource Calendar. We set GarQ with $\delta = 5$ minutes, and a fixed interval length of 30 days. Finally, we simulate a system with 64 homogeneous compute nodes, i.e. $maxCN = 64$.

6.4.1 Simulation Setup

We use a workload trace of the San Diego Supercomputer Center (SDSC) Blue Horizon obtained from the Parallel Workload Archive [49]. This trace is chosen because it represents a large number of jobs and contains a mixture of single and parallel jobs. Note that we only simulate the first 2-weeks period of the trace, which is approximately 3200 jobs, since the original trace was recorded over a two-year period. We selected 30% of these jobs to use reservation. Few modifications have also been made to this trace, as mentioned below:

- If a job requires more than the total number of nodes of a resource, we set this job to $maxCN$.
- A request's start time is rounded up to the nearest 5-minute time interval. For example, if a job request starts at time 01:03:05 (hh:mm:ss), then it will be rounded to time 01:05:00.

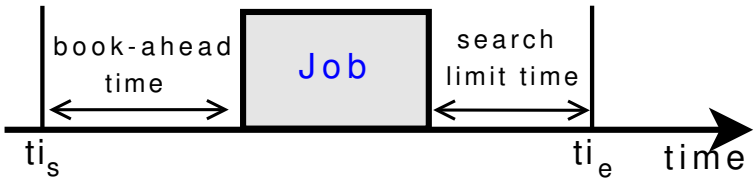


Figure 6.3: Degree of flexibility of a reservation query.

- A job duration time is within the range of 5 minutes to 28 days. We limit the maximum duration time to prevent overlapping reservations from different months. Hence, the data structure can be reused and built only once.

For the evaluation, we are investigating: *(i)* the effects of having the elastic reservation model compared to the rigid model. These effects include the average resource utilization, and the total number of rejections by the system; *(ii)* the impact of elastic and rigid models to non-reserved jobs, where we measure the average waiting time these jobs spent in the Job Queue; and *(iii)* the degree of flexibility given to the elastic model, where we vary the $[ti_s,\ ti_e]$ interval of a reservation query, by using the following parameters:

- book-ahead time, b_t, where it denotes the booking time *prior* to the job's starting time t_s (as stated in the SDSC trace), as shown in Figure 6.3. In the experiment, we use $b_t \in \{1, 5, 10\}$ hours.

- search limit time, sl_t, where it denotes the time *appended* at the end of the job, as shown in Figure 6.3. In the experiment, we use $sl_t \in \{0, 1, 2, 4, 6, 8, 10, 12\}$ hours.

6.4.2 User's Selection Policy

As mentioned earlier, the user submits a reservation query to a resource. Then, he/she will receive a list of offers, $offerList[\ ]$, from the resource. Algorithm 9 shows the user's selection policy in choosing the best offer (line 1–9).

In Algorithm 9, the user is willing to accept an offer by reducing the initial *dur* and *numCN* objectives, by up to a half or δ and 1 respectively (line 1–2). Therefore, the list needs to be sorted in decreasing order based on the duration time, i.e. from the longest to the shortest duration time (line 3). Then, each offer in the list is checked against the *minDur* and *minCN* objectives (line 4–9). If a suitable offer is found, the user will place

Algorithm 9: The selection policy of a user.

Input: $offerList[\,]$ or a list of offers

```
1  minDur ← max(dur / 2, δ);
2  minCN ← max(numCN / 2, 1);
3  offerList[ ] ← sort_decreasing(offerList[ ]);        // based on the duration time
4  for (i = 0) to (size − 1) do
5  |  offer ← offerList[i];
6  |  if is_suitable(offer, minCN, minDur) == true then
7  |  |  return offer;            // found a suitable offer, so make a reservation
8  |  end
9  end
10 return φ;                                            // no suitable offers found
```

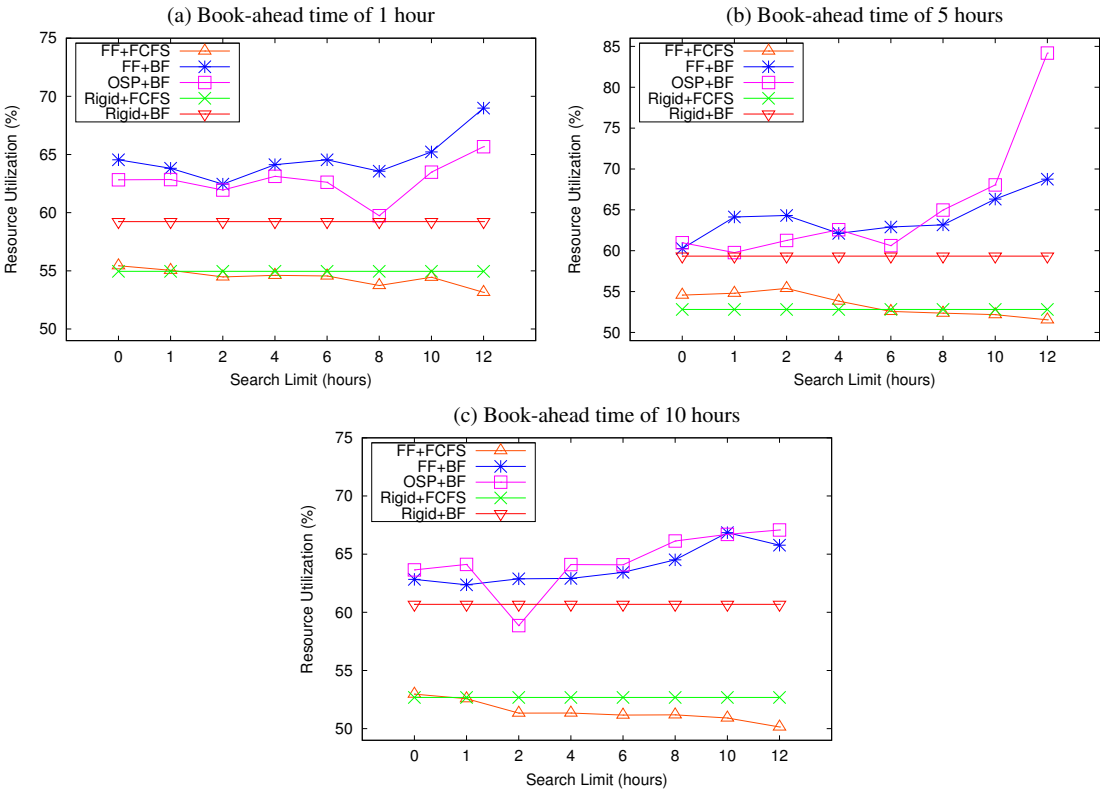


Figure 6.4: Average resource utilization.

a reservation on this offer (line 7). Otherwise, the user ignores the given offers (line 10). Note that this selection policy is overly simplified and might not be feasible in real Grid applications. However, we do this in order to demonstrate the elasticity of the proposed model and the effectiveness of the OSP algorithm.

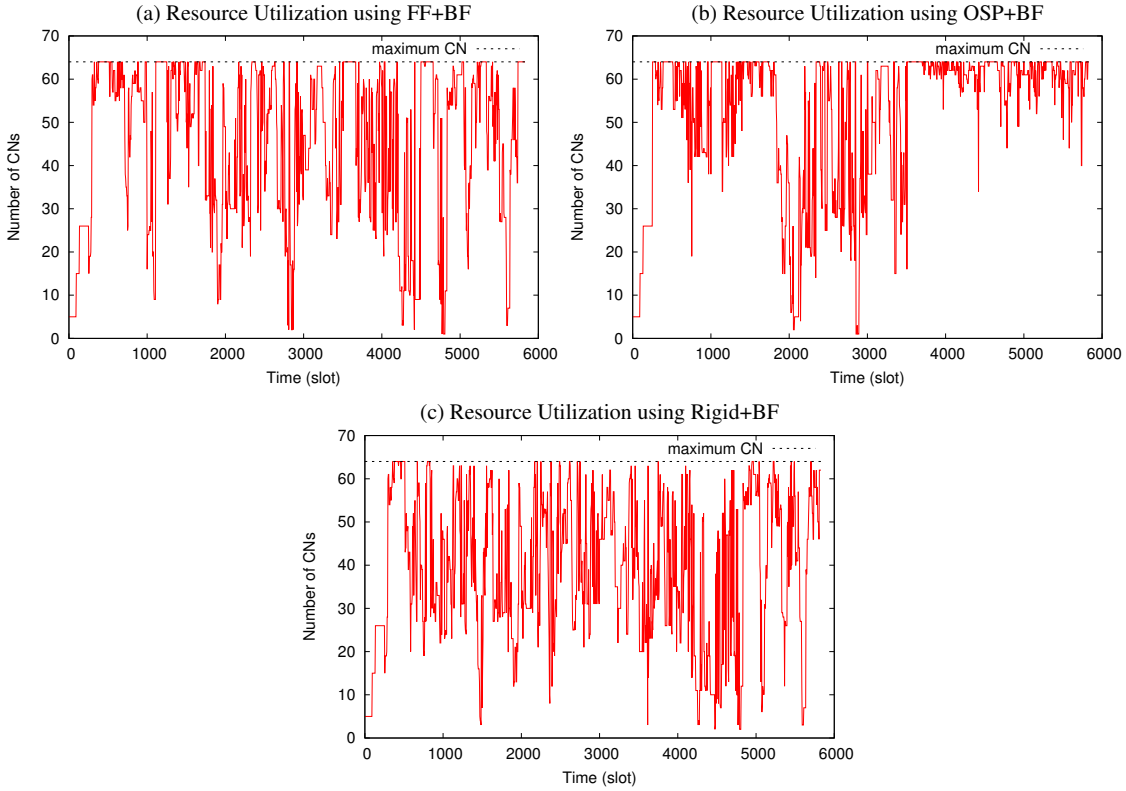


Figure 6.5: Total number of busy CNs over a two-week period, with $\delta = 5$ minutes, $b_t = 5$ hours and $sl_t = 8$ hours.

6.4.3 Results

Figure 6.4 shows the effects of having reservations on the resource utilization. The result of this figure is also influenced by the choice of a good scheduling policy, where BF manages to perform much better than FCFS in all cases, by more than 4%. This can be shown by comparing $Rigid + FCFS$ with $Rigid + BF$, and $FF + FCFS$ with $FF + BF$. For the two Rigid algorithms, the gap between FCFS and FF is 4.3%, 6.5% and 8% for b_t of 1, 5, and 10 hours respectively. For the two FF algorithms, the gap is even bigger, i.e. 11% on average of all b_t results, ranging from 5.7% ($sl_t = 0$) to more than 15% ($sl_t = 12$).

Having a degree of flexibility in the reservation requests allows an additional improvement in the resource utilization, as depicted in Figure 6.4. The elastic model (i.e. $OSP + BF$) improves the resource utilization by 4.39% on average compared to the rigid model (i.e. $Rigid + BF$). Figure 6.4 also shows that the resource utilization stays constant for both $Rigid + FCFS$ and $Rigid + BF$, since because they treat the input parameters as hard constraints. Thus, the b_t and sl_t values do not have any effects on these Rigid

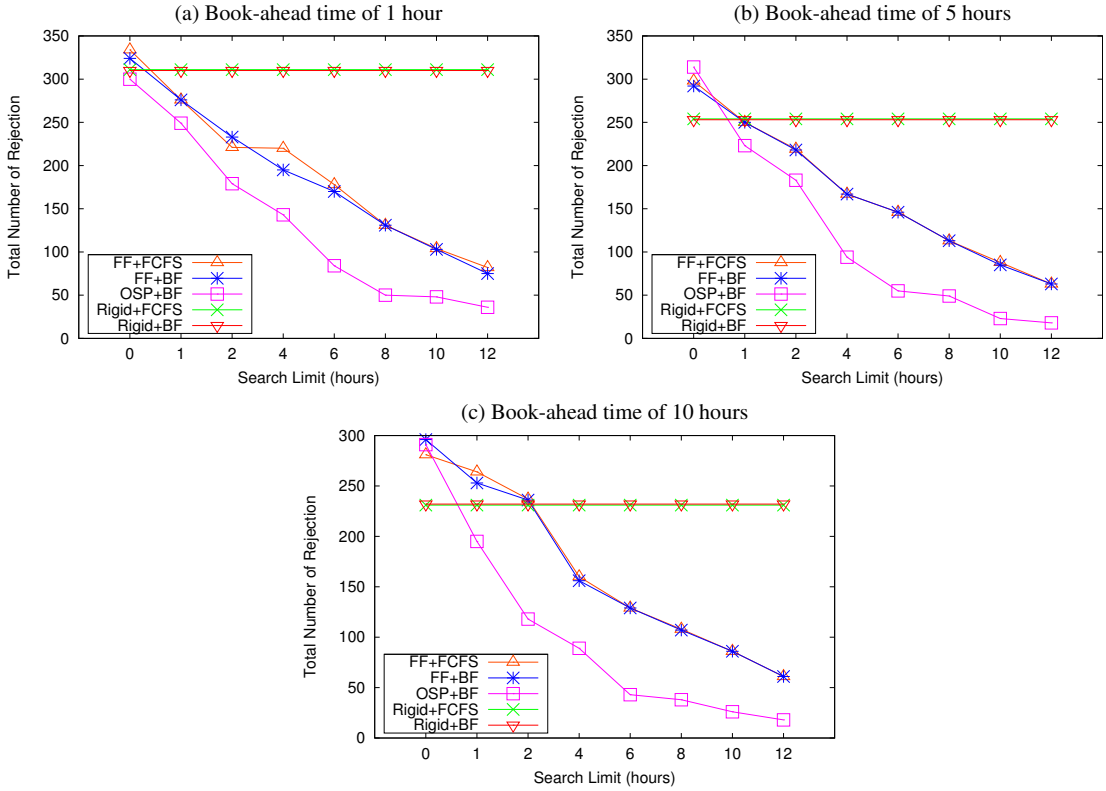


Figure 6.6: Total Number of Rejection (lower number is better).

algorithms.

In Figure 6.4 (a), $OSP + BF$ behaves slightly worse to $FF + BF$, since b_t is too small to make any improvements for the resource utilization. However, when b_t is larger and $sl_t \geq 6$ hours in Figure 6.4 (b) and (c), the performance of $OSP + BF$ is improving, and performing better than $FF + BF$ by 2.5% on average.

Figure 6.5 looks at the resource utilization in more details, as it shows the total consumption of nodes for the entire duration. $FF + BF$ and $Rigid + FCFS$, as shown in Figure 6.5 (a) and (c) respectively, fluctuate frequently throughout. This condition can be interpreted as having too many fragmentations or idle time gaps in the system. In contrast, $OSP + BF$ manages fragmentations better since reserved jobs are assigned to slots within a local minima of free nodes, as displayed in Figure 6.5 (b). Thus, in the best scenario, all nodes are busy or close to full, while at the same, leaving some empty nodes available at different time periods. As a result, reserved and non-reserved jobs that require many nodes have a lower probability of being rejected compared to the FF and Rigid algorithms on average, as shown in Figure 6.6.

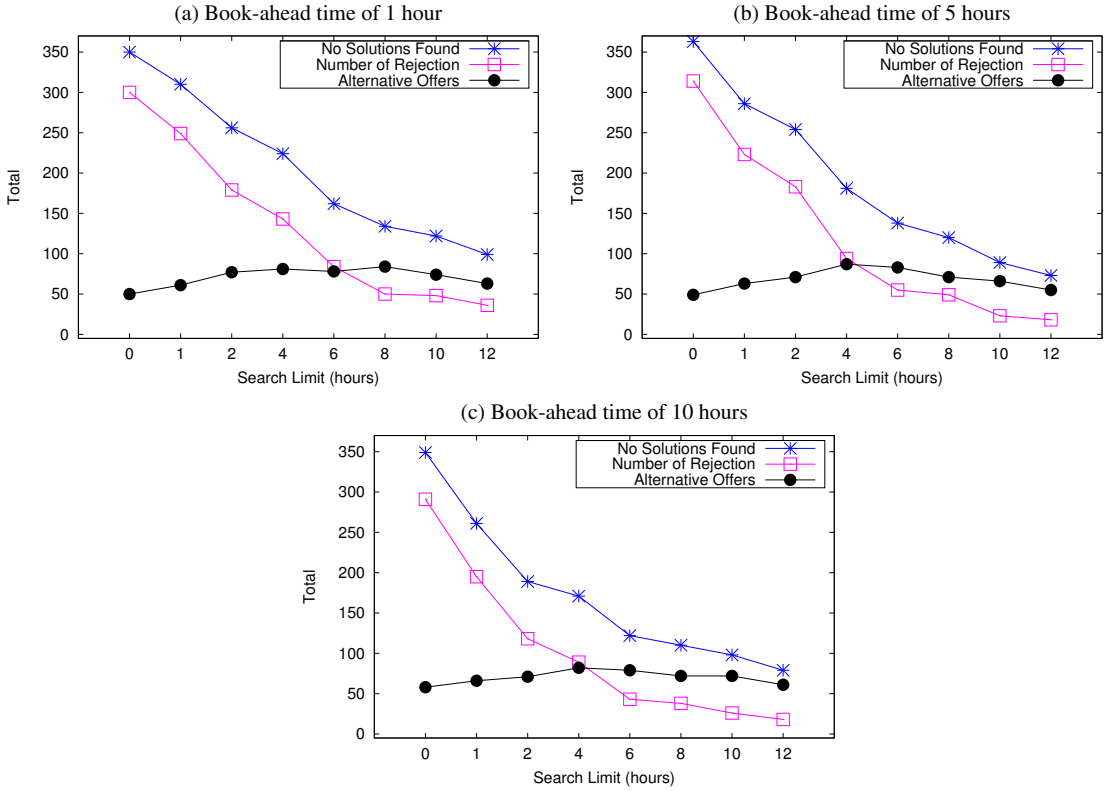


Figure 6.7: Degree of flexibility in reserving AR jobs for the $OSP + BF$ algorithm.

With the elastic model, users can self-select which alternative offers to choose, if no solution is found. Thus, they can reduce the initial $numCN$ and/or dur values according to Algorithm 9, and select the most suitable offer from the list. Figure 6.6 shows that, as sl_t increases, $OSP + BF$ has the lowest number of rejection. For $sl_t = 0$ in Figure 6.6 (b) and (c), $OSP + BF$ performs worse than the Rigid algorithms since it does not allow to search for alternative solutions at later times. However, as sl_t increases, $OSP + BF$ manages to reduce the number of rejections by at least 12% ($sl_t = 1$) to 88% ($sl_t = 12$) compared to $Rigid{+}FCFS$, as shown in Figure 6.6. On average, the elastic model reduces the number of rejections by 54.88% and 41.67% compared to the Rigid and FF algorithms, respectively.

Figure 6.7 also shows the importance of sl_t for the elastic model. As sl_t increases, $OSP + BF$ manages to find solutions that satisfy the given parameters. This figure also shows that by allowing users to select an alternative offer if no solutions are found, it reduces the total number of rejection, by at least 13.5% ($sl_t = 0$) to 63.6% ($sl_t = 12$).

Finally, Figure 6.8 shows the impact of reservation for non-reserved jobs, in terms of

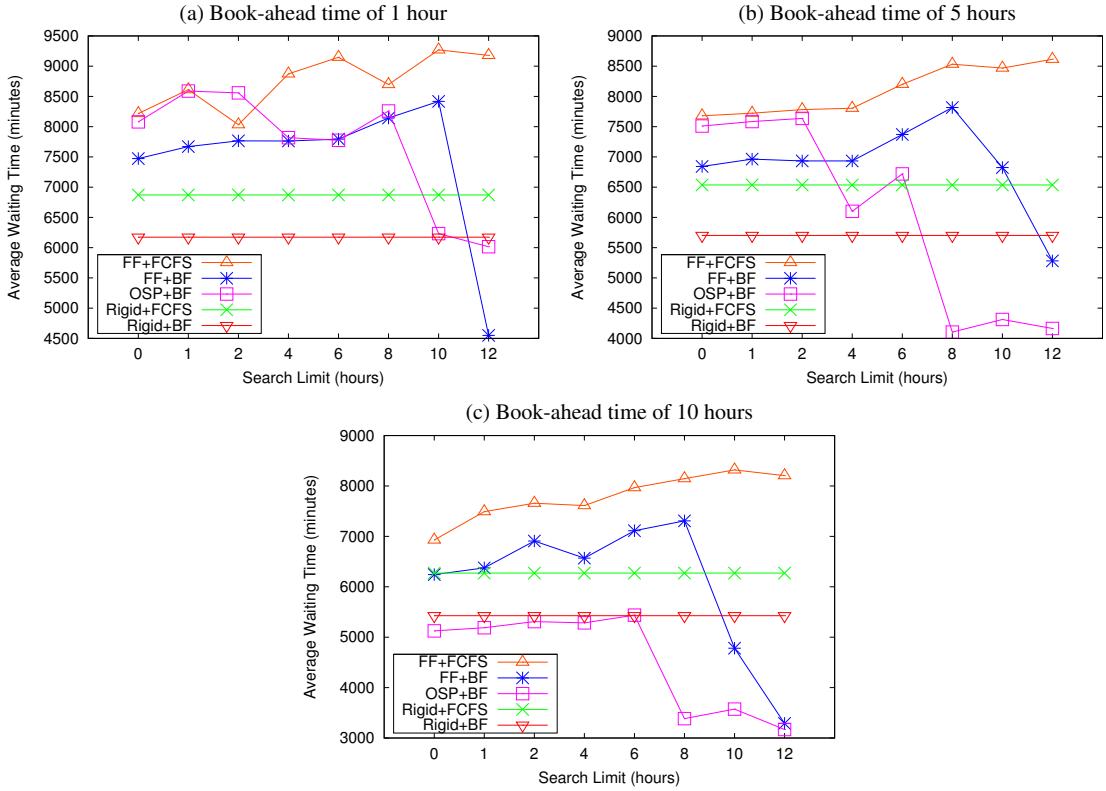


Figure 6.8: Average waiting time for non-reserved jobs (lower number is better).

the average waiting time in the Job Queue. When $b_t = 1$, the Rigid algorithms have the lowest impact on average, as shown in Figure 6.8 (a). This is because they reject the most reservations, as mentioned previously. For $OSP+BF$, the impact is worse when a request has a short time interval, e.g. $sl_t \leq 2$ in Figure 6.8 (a) and (b), due to not enough room for flexibility. However, for the same sl_t, as b_t becomes larger, $OSP + BF$ manages to minimize the waiting time by at least 22% on average. Eventually, $OSP + BF$ performs better than the Rigid algorithms for $b_t = 10$, as highlighted in Figure 6.8 (c). Note that, this result is influenced by the frequency of jobs arrival rate and the choice of a good scheduling policy, where BF performs better than FCFS.

6.5 Related Work

Strip packing is a generalization of bin packing [83]. Bin packing is an $\mathcal{NP}$-hard problem, where it aims to minimize the number of bins used to store a set of objects of different sizes. Many variants of bin packing have been proposed by several researchers [11, 115].

A flexible method for reserving jobs in Grids has been presented [26, 69, 76], where they talk about extending the reservation time interval or window in order to increase the success rate. However, they do no provide alternative offers if the reservation is rejected. On the other hand, the work done by [116, 124] provides this important functionality.

The fuzzy model introduced by Roeblitz et al. [116] provides a set of parameters when requesting a reservation, and applies speedup models for finding the alternative solutions. Moreover, their model requires additional input conditions, such as the gap between two consecutive time slots and maximum number of time slots. However, no optimization on the resource utilization is considered in their model. In addition, our model aims to reduce fragmentations, hence, it does not require to specify the gap between time slots.

The model proposed by Siddiqui et al. [124] uses a 3-layered negotiation protocol, where the allocation layer deals with flexible reservations on a particular Grid resource. In this layer, the authors also used the strip packing method. However, the resources are dynamically partitioned into different shelves based on demands or needs, where each shelf is associated with a fixed time length, number of CNs and cost. Thus, the reservation request is placed or offered into an adjacent shelf that is more suitable. In contrast, our model does not need different shelves with variable length, since we use a time-slotted data structure, based on a fixed time interval δ. Therefore, our approach is focusing more on utilizing the compute nodes for each time slot in the data structure.

In networks, Naiksatam and Figueira [100] propose an elastic model for bandwidth reservations, by partitioning the network capacity into slots. Then, they present a heuristic algorithm, Squeeze In Stretch Out (SISO), to schedule bandwidth reservations. Each reservation is associated with a minimum and a maximum number of bandwidth slots for a guarantee QoS. Thus, SISO can increase (squeeze in) or decrease (stretch out) the allocated slots of each reservation over the time period, in order to increase the overall bandwidth utilization. However, this approach is not feasible, since in our model the compute nodes are fully dedicated to executing one reservation at a time (a space-shared mode). Thus, they can not be shared with other reservations or jobs.

In a real-time system, Kim [77] extends the DSRT scheduling system to provide alternative offers if a request is rejected, as described in Section 2.2.2. In addition, the CPU broker of the DSRT system allows the users to specify what to expect in case their

reservations finish early or late, as mentioned previously. We will consider this feature as a future work.

6.6 Summary

This chapter provides a case for an elastic reservation model, where users can *self-select* or choose the best option in reserving their jobs, according to their Quality of Service (QoS) needs, such as deadline and budget. In this model, each Grid system has a Reservation System and a Resource Calendar. The Reservation System is responsible for handling reservation queries and requests, whereas the Resource Calendar is responsible for storing and updating information about resource availability as time progresses. For the Reservation System, the model adapts an on-line strip packing (OSP) algorithm. For the Resource Calendar, the model uses GarQ, as explained in Chapter 5.

The OSP algorithm considers the duration and number of required compute nodes as soft constraints for a given reservation query. Thus, it aims to find a solution or alternative offers within the given time interval for users to choose themselves. Rather than giving the first available empty slots to users, the OSP algorithm plans ahead and targets at a slot which represents a local minima, based on the remaining number of available nodes recorded in GarQ. In the best case scenario, all nodes at this slot become busy. As a consequence, other slots can be used to run jobs that require more than one node. Thus, the OSP algorithm also aims to reduce fragmentations or idle time gap caused by having reservations in the system.

Having a degree of flexibility in the reservation requests allows an improvement in the resource utilization. Results show that the elastic model improves the resource utilization by 4.39% on average compared to the rigid model. In addition, the elastic model reduces the number of rejections by 54.88% on average compared to the rigid model. The results also show that by allowing users to select an alternative offer if no solutions are found, the OSP algorithm reduces the total number of rejection by around 13.5% – 63.6%. Note that the rigid model treats all the request parameters as hard constraints. Therefore, if no solution is found, then the rigid model will reject such requests.

The challenging issue of adopting advance reservation in existing Grid systems is its

impact in increasing the waiting times of local jobs in the queue. As expected, results show that the rigid model has a minimal impact on the average waiting time, as it did not accept too many reservations. However, the elastic model performs better as the reservation requests become more flexible. The results show that the elastic model improves its performance by 22% on average. The elastic model performs better than the rigid model for requests with a book-ahead time of 10 hours.

In addition, there are several issues need to be addressed by the Grid systems, such as calculating reservation price, increasing resource revenue, and regulating supply and demand. In the next chapter, we propose the use of Revenue Management to address these issues.

Chapter 7

Revenue Management, Overbooking and Reservation Pricing

This chapter proposes the use of Revenue Management to determine the pricing of reservations in order to increase the resource revenue, and to regulate supply and demand. In addition, this chapter introduces the concept of *overbooking* to protect the resource against unexpected cancellations and no-shows of reservations.

7.1 Introduction

Buyya et al. [21] introduced a Grid economy concept that provides a mechanism for regulating supply and demand, and calculates pricing policies based on these criteria. With this concept, it offers an incentive for resource owners to join the Grid, and encourages users to utilize resources optimally and effectively.

A study by Smith et al. [132] showed that by providing advance reservation (AR) in Grid systems, it increases waiting times of applications in the queue by up to 37% with backfilling. This study was conducted, without using any economy models, by selecting 20% of applications using reservations on across different workload models. The finding implies that without economy models or any set of policies, the systems accept reservations based on a first come first serve basis and subject to availability. It also means that these reservations are treated similarly to high priority jobs in a local queue. Therefore,

regulating supply and demand is an important issue in advance reservation.

Revenue Management (RM) can be an answer for the aforementioned problems. The main objective of RM is to maximize profits by providing the right price for every product to different customers, and periodically update the prices in response to market demands [111]. Therefore, a resource provider can apply RM techniques to *shift demands* requested by budget conscious users to off-peak periods as an example. Hence, more resources are available for users with tight deadlines in peak periods who are willing to pay more for the privilege. As a result, the resource provider gains more revenue, and allocates available nodes to applications that are highly valued by the users in this scenario. So far, RM techniques have been widely adopted in various industries, such as airlines, hotels, and car rentals [92].

7.2 Revenue Management Techniques and Strategy

Revenue management (RM) is applicable when the following requirements are met [111]:

- Capacity is limited and immediately perishable. For example, an empty hotel room of today cannot be stored to satisfy future demand.
- Customers book capacity ahead of time to guarantee its availability when they need to consume it.
- Seller manages a set of fare classes and updates their availability based on market demands.

From the above criteria, RM is suitable in determining the pricing of reservations in Grids, as computing powers can be considered perishable. To successfully adapt RM, a resource provider needs to have an initial *strategy*, establishes a system that handles *bookings* and updates its *tactics* periodically based on demands [111]. These aspects are discussed next.

7.2.1 Market Segmentation

This is an initial step of RM that identifies different customer segments for a product, and applies different pricing to each of them. The resource provider only needs to come up

Table 7.1: An example of market segmentation in Grids for reserving jobs.

Class	User Category	Restrictions
1	Premium	none
2	Business	same VO, allow cancellation
3	Budget	same VO, non-refundable, only for a limited number of CNs

Table 7.2: Characteristics of different users.

Budget User	Business and Premium User
Relaxed deadline	Tight deadline
Run longer jobs	Run short/medium jobs
Highly price sensitive	Less price sensitive
Book earlier	Book later
More flexible	Less flexible
More accepting of restrictions	Less accepting

with a strategy quarterly or annually. Note that a product in the Grid context means a resource requested by users in advance.

The airlines industry is a well-known example that segments customers and offers them different fare classes based on when they book their flights prior to departure times. Each *fare class* is a combination of a price and a set of restrictions on who can purchase the product and when. For example, a customer that books a flight one day prior to a departure time can be identified as a business customer. The airline knows from historical data that business customers are less flexible to changes and less price sensitive than leisure customers who book a week before. Therefore, the airline can sell a higher price to business customers compared to leisure customers for seats in a same flight.

In Grids, resources can be part of one or more *virtual organizations* (VOs). The concept of a VO allows users and institutions to gain access to their accumulated pool of resources to run applications from a specific field [54], such as high-energy physics or aerospace design. Table 7.1 shows an example of market segmentation in Grids, where we classify users into three classes, i.e. *Premium*, *Business* and *Budget*. The classifications are based on user VO domains and a set of conditions or restrictions imposed on each user category. In addition, we profile users according to their Quality of Service (QoS) requirements (e.g. deadline and cost) and job patterns (e.g. job size and time of bookings), as depicted in Table 7.2.

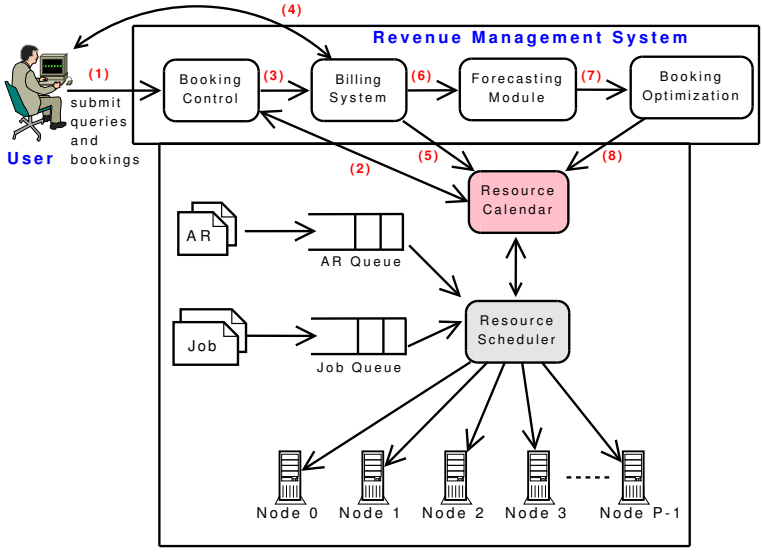


Figure 7.1: Revenue Management System as part of a Grid resource.

7.2.2 Price Differentiation

Once users' classifications and profiling are identified, restrictions can be introduced to create virtual products oriented toward different market segments to make additional profits. As an example, products for the *Budget* users have many restrictions, as shown in Table 7.1, that make them unsuitable and unavailable to users with tight deadlines and from different VOs respectively. As a result, an *inferior* product can be sold to a more price-sensitive segment of the market [111]. Therefore, the resource provider can set prices for the same product to be: $p_1 > p_2 > p_3$, where p_1 denotes the price paid by the *Premium* (class 1) users and so on. This practice is commonly known in the economics literature as *price differentiation* or *discrimination*.

The main advantage of this approach is that these prices can be adjusted dynamically based on demands, since Grid resources are limited. Hence, by increasing the price to all classes during peak periods, it can shift some demands from the *Budget* users to off-peak periods. As a result, more resources are available for reservations for both the *Premium* and *Business* users.

7.3 Revenue Management System

Figure 7.1 shows how Revenue Management System (RMS) can be integrated into the existing elastic Grid reservation-based system, which was discussed in Chapter 6. With the adoption of the RMS, the functionalities of the Reservation System are integrated into the *Booking Control* (BC). Thus, the BC is now responsible for handling users queries and bookings (step 1). This is done by consulting and checking booking limits in the Resource Calendar (step 2).

A *booking limit* (b) is the maximum number of nodes that may be reserved at each fare class. Therefore, each slot in the data structure, as explained in Section 5.3, is modified to contain b_1, b_2, and b_3 denoting the booking limit for class 1, 2 and 3 respectively.

Once the query yields a list of options, the *Billing System* (BS) calculates a fare class for each of them (step 3). Then, the BS sends this information to the user (step 4). The BS also handles the user payment and confirms his/her booking by submitting this information to the Resource Calendar (step 5).

Forecasting Module (FM) is responsible for generating and updating forecasts of demands in the future. Initially, the forecast can be done about two to three weeks prior to an opening of bookings. Then the FM updates this forecast frequently as bookings and cancellations are received over time from the BS (step 6).

These forecasts are then used as inputs by the *Booking Optimization* to re-generate booking limits for each user class in the Resource Calendar (step 7 and 8). Hence, if the demands are deemed to be low, the booking limit for the *Budget* users is set to a higher number in order to increase the existing capacity. Forecasting and optimization will be discussed next.

7.4 Revenue Management Tactics

RM tactics are used in a daily operational planning to calculate and update booking limits. For these tactics, we assume that class 3 (*Budget*) users reserve *before* class 2 users *before* class 1 users, as shown in Figure 7.2. This assumption is used so that once a booking limit for class 3, b_3, is reached, then users will be offered a fare class of the next one, i.e. class 2, and so on.

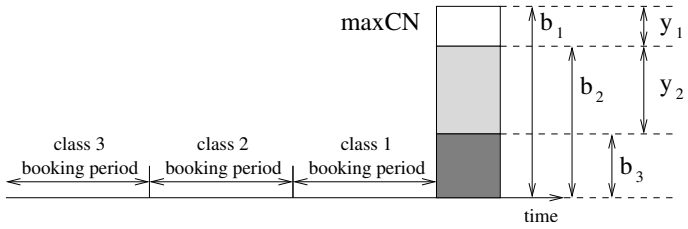


Figure 7.2: Protection levels (y_1, y_2) and nested booking limits (b_1, b_2, b_3) for each slot.

7.4.1 Protection Levels and Nested Booking Limits

When an initial demand is generated, the *Forecasting Module* sets protection levels, y_1 and y_2 for class 1 and 2 respectively. A *protection level* (y) is required in order to make some CNs available for business and premium users that might book later in time, as shown in Figure 7.2.

In order to prevent high-fare bookings are being rejected in favor of budget ones, a nested approach is used to determine b_i, where b_i denotes the booking limit for class i, as shown in Figure 7.2. With this approach, the booking limits are always non-increasing, i.e. $b_1 \geq b_2 \geq b_3$. In addition, every class has access to all of the bookings available to lower classes. Hence, b_1 denotes the maximum number of CNs to be reserved.

7.4.2 Calculating Booking Limit for Two-Fare Class Users

Let us first consider a two-class user problem for a given capacity C for simplicity, where h denotes a higher class and l denotes a lower class. Let p_i denotes the price of class i. Since the price of a higher class is more expensive than that of a lower class, as mentioned in Section 7.2.2, it follows that $p_h > p_l$.

We assume that a cumulative distribution function of class i's demand is given by $F_i(x)$, because the analysis is based on forecasting future bookings [92]. Thus, $F_i(x)$ is the probability that the demand of class i user is less than or equal to x.

We assume that the current booking limit for the lower class is $b_l - 1$. The expected revenue (E) can be changed by $IR(b_l)$, where $IR(b_l)$ denotes the increase of b_l by 1. In addition, E depends on the demand of the lower-class users (d_l). If $d_l \leq (b_l - 1)$, then the expected revenue is the same. However, if $d_l > (b_l - 1)$, then the revenue depends on d_h.

When E relies on the demand of the higher-class users (d_h), we encounter two pos-

sibilities. If $d_h \leq (C - b_l)$, then the revenue can be increased by a minimum of p_l. On the contrary, if $d_h > (C - b_l)$, the resource provider will lose by at least $(p_h - p_l)$. The expected revenue increase from $b_l - 1$ to b_l is defined by the following [111]:

$$\begin{aligned} E[IR(b_l)] &= (1 - F_l(b_l - 1)) \times \{F_h(C-b_l)p_l - (1-F_h(C-b_l))(p_h-p_l)\} \\ &= (1 - F_l(b_l - 1))\{p_l - (1 - F_h(C - b_l))p_h\} \end{aligned}$$

The algorithm to calculate b_l is shown in Algorithm 10, where it starts from zero and keeps incrementing until the increased expected revenue becomes zero or negative. As a result of Algorithm 10, the protection level of a higher class is also determined by $C - b_l$.

Algorithm 10: BookingLimit $(C,\ p_h,\ p_l,\ F_h)$

```
b_l ← 0;
while b_l < C do
    b_l ← b_l + 1 ;
    E[IR(b_l)] ← (1−F_l(b_l − 1)){p_l−(1−F_h(C − b_l))p_h} ;
    if E[IR(b_l)] ≤ 0 then return b_l − 1 ;
end
return b_l ;
```

7.4.3 Capacity Allocation in Three-Fare Class Users

The capacity allocation problem in RM is to decide the booking limit for each class user, in order to maximize the overall expected total revenue. If too many CNs are allocated to lower-class users, we may lose a chance to earn more revenue from accepting future bookings from higher-class users, e.g. in peak periods. On the contrary, an insufficient quota for the lower-class users, may lead to a lower resource utilization and revenue, e.g. in off-peak periods. Thus, determining an appropriate capacity allocation to each user class at different time periods is an important factor in RM.

Let us consider the capacity allocation problem of three classes in the RMS. We use an expected marginal seat revenue (EMSR) heuristic [10] to determine the booking limits of three classes, as shown in Algorithm 11. In order to determine b_3, the protection levels of class 1 and 2 need to be calculated first, as shown in Algorithm 11. Then, b_2 can be found by using the two-class problem with $C = maxCN - b_3$.

Algorithm 11: Capacity Allocation in Three-fare Class Users.

1 $y_1 \leftarrow maxCN - BookingLimit\ (maxCN,\ p_1,\ p_3,\ F_1)$;
2 $y_2 \leftarrow maxCN - BookingLimit\ (maxCN,\ p_2,\ p_3,\ F_2)$;
3 $b_3 \leftarrow \max(0,\ maxCN - y_1 - y_2)$;
4 $b_2 \leftarrow b_3 + BookingLimit\ (maxCN\ - b_3,\ p_1,\ p_2,\ F_1)$;
5 $b_1 \leftarrow maxCN$;

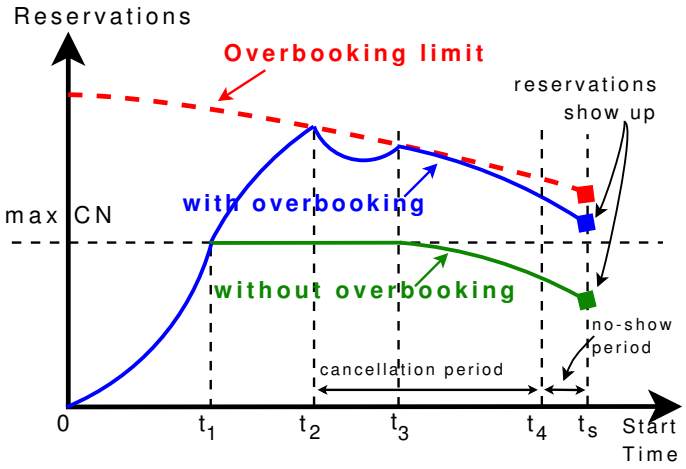


Figure 7.3: An example of total number of reservations with and without overbooking.

7.5 Overbooking

Once users book a certain amount of CNs, the resource provider expects them to submit their jobs before reservations start. However, in reality, users may cancel their reservations before starting time or by not submitting at all (*no-show*), due to reasons such as resource or network failures on the other end. Thus, the resource provider has no choice but to reject bookings from potential users, who are willing to pay for a higher price and committing to use the resource during a period of high demands for example. As a result, the resource provider is faced with a prospect of loss of income and lower system utilization.

Overbooking offers a solution for the above problem, by allowing the resource provider to accept more reservations than the capacity. Hence, it can be effectively used to minimize the loss of revenue [92, 111]. However, the challenging issues in using overbooking are determining the appropriate number of excess reservations, minimizing total compensation costs, addressing legal and regulatory issues, and dealing with market acceptance, especially the *ill-will* or negative effects from users who have been denied access [142]. In this section, we only consider the first two issues of overbooking.

Figure 7.3 illustrates an example on how overbooking can protect a resource provider against unanticipated cancellations and no-shows. We define a *cancellation* as a reservation that is terminated by a user *before* the service or starting time t_s, as shown in Figure 7.3. Moreover, we describe a *no-show* as a reservation that *fails* to arrive and run on the resource on t_s (without a cancellation notice).

By setting the overbooking limit ob to be greater than $maxCN$, the resource provider can still accept more reservations (after t_1) until total number of reservations tot_{AR} equals to ob (on t_2 and t_3), as shown in Figure 7.3. In contrast, a resource without overbooking has to deny potential reservations starting from t_1, since the capacity is full.

The overbooking limit itself needs to be updated and evaluated frequently as t_s approaches. Thus, as tot_{AR} increases, ob decreases, as shown in Figure 7.3. Then, the resource provider takes an advantage of the cancellation and no-show periods to reduce tot_{AR}. In the best-case scenario, the resource may not need to deny any excess reservations due to a large number of no-shows. In the end, on t_s, a resource with overbooking yields more reservations that show up than without overbooking. Note that, if $tot_{AR} > maxCN$, we address this problem by introducing a compensation scheme. More details on this scheme is discussed in Sections 7.5.2 and 7.6.3.

In this section, we adopt several *static* overbooking policies, introduced in the RM literature [111, 142], into our work. These static policies only calculate the ideal overbooking limit periodically prior to t_s, when the state and probabilities change over time. Thus, we assume the following things:

- Cancellations and no-shows are independent of the number of total bookings.
- The probability of a cancellation is Markovian, i.e. it only depends on the current time.
- No-shows are treated as cancellations on t_s. Hence, we can define $q(t)$ as a show rate or a probability that reservations show up from the time remaining until t_s.

7.5.1 A Probability-based Policy

This is a simple overbooking policy, where ob is determined statistically based on the probability of shows. Equation 7.1 determines the overbooking limit at time t. For exam-

ple, if $maxCN = 100$ and $q(t) = 0.80$, then the amount of overbooking capacity is 125. Therefore, the lower the probability of shows, the higher the overbooking limit becomes.

$$ob = \frac{maxCN}{q(t)} \tag{7.1}$$

7.5.2 A Risk-based Policy

A risk-based policy aims to balance the expected cost of denied service with the revenue by accepting more bookings. The cost of denied service refers to the compensation money given to users who got rejected or *bumped* at the service time. This cost of denied service is denoted as $cost_{ds}$ and is usually higher than the reservation price p. Thus, a risk analysis is required in order to calculate a threshold at which the overbooking is allowed.

For computing the threshold, we need to find out the probability distribution of users demand and number of shows. Let $A(x)$ denotes the probability that the demand of users is less than or equal to x, where x denotes the number of bookings. Moreover, we define $F_x(y)$ as the probability that the number of bookings that will show up at the time of service is less than or equal to y.

We derive the show distribution $F_x(y)$ at time t, under the assumption that each customer's showing probabilities are independent. The show probability of each customer at time t is denoted as q. Then, the number of shows, as investigated by Thompson [145], follows a binomial distribution with the cumulative distribution function:

$$F_x(y) = \sum_{k=0}^{y} \binom{x}{k} q^k \ (1-q)^{x-k} \tag{7.2}$$

Let us assume that the current booking limit and capacity are b and C, respectively. Then, we derive the expected revenue change by increasing the booking limit from b to $b+1$. By doing this, we are faced with three possible cases:

1. $demand < b+1$, which means there are no changes in the forecasted revenue.

2. $demand \geq b+1$ and the number of shows $\leq C$. Since the resource provider can serve users at t_s, the profit of p is obtained by accepting an additional reservation.

3. $demand \geq b+1$ and the number of shows $> C$, which means the resource provider has

to deny one user with a compensation cost. As a result, there is a loss of $cost_{ds} - p$, where $cost_{ds} > p$.

Thus, we can derive the expected revenue change by increasing the booking limit from b to $b+1$ as follows.

$$\begin{aligned} E[R|b+1] - E[R|b] &= (1 - A(b))\{pF_{b+1}(C) + (p - cost_{ds})(1 - F_{b+1}(C))\} \\ &= (1 - A(b))\{p - cost_{ds}(1 - F_{b+1}(C))\} \end{aligned} \tag{7.3}$$

Algorithm 12: Overbooking Limit using a Risk Policy

```
1 ob ← C;
2 IR ← (1 − A(ob)){p − cost_ds(1 − F_{ob+1}(C)) ;
3 while IR > 0 do
4 |   ob ← ob + 1 ;
5 |   IR ← (1 − A(ob)){p − cost_ds(1 − F_{ob+1}(C)) ;
6 end
7 return ob ;
```

As long as the expected revenue change is greater than zero, the overbooking limit can be increased, as shown in Algorithm 12. The booking limit ob starts from the maximum capacity C (line 1), and is incremented until the expected revenue change becomes zero or negative (line 3).

In multiple fare classes, the increased revenue from having an additional booking can not be easily calculated. Therefore, a suitable approach is to determine a weighted average price $\hat{p}$, based on the mean demands μ in each user class [111]. More specifically,

$$\hat{p} = \sum_{i=0}^{n} \mu_i p_i \tag{7.4}$$

7.5.3 A Service-Level Policy

Although the risk-based policy enhances the expected revenue of the resource, users who got denied at the service time, tend to submit their jobs to other resources in the future. Thus, by using the risk-based policy, a resource may lose some of these users in the long term. Moreover, this policy may increase the negative impact of overbooking towards users' satisfaction.

A service-level policy addresses the above issues by defining a specified level or fraction

of denied users. For example, American Airlines and United Airlines have an involuntary denied boarding (DB) ratio of 0.84 and 0.51 per 10,000 passengers respectively, due to oversales in 2006 [104]. The data were taken from flights within and originated in the United States. With the service-level policy, the airlines may set a threshold of involuntary DB ratio to be 0.50 as an example. Accordingly, the airlines could determine the overbooking limit based on this threshold.

Suppose that the number of shows for a given x bookings is denoted as $B(x)$. Then, the service level of x bookings, $s(x)$ is defined by Equation 7.5, where $(B(x) - C)^+ = \max(0, B(x) - C)$. The equation implies the fraction of the expected denied service over the expected number of shows.

$$s(x) = \frac{E[(B(x) - C)^+]}{E[B(x)]} \tag{7.5}$$

If we use a binomial distribution for show demands, then the service level of x bookings can be derived as follows. The probability mass function of a binomial distribution:

$$\begin{aligned} P_x(k) &= P(B(x) = k) \\ &= \binom{x}{k} q^k (1-q)^{x-k}, \quad k = 0,\ 1,\ \ldots,\ x \end{aligned} \tag{7.6}$$

with mean $E[B(x)] = xq$ and variance $Var(B(x)) = xq(1-q)$. According to Talluri and Ryzin [142], Equation 7.5 can further be refined to

$$s(x) = \frac{\sum_{k=C+1}^{x} (k - C)\ P_x(k)}{xq} \tag{7.7}$$

Thus, substituting Equation 7.6 into Equation 7.7, we obtain

$$s(x) = \frac{1}{xq} \times \sum_{k=C+1}^{x} (k - C) \binom{x}{k} q^k (1-q)^{x-k} \tag{7.8}$$

For a given service level *ds_threshold*, the overbooking limit for this policy is calculated in Algorithm 13. Initially, the overbooking limit *ob* starts from the maximum capacity C (line 1), and is incremented until $s(x)$ equals to or less than *ds_threshold* (line 3).

Algorithm 13: Overbooking Limit using Service Policy

$ob \leftarrow C$;
$s(x) \leftarrow E[(B(x) - C)^+]$ / $E[B(x)]$;
while $s(x) \leq ds\_threshold$ **do**
    $ob \leftarrow ob + 1$;
    $s(x) \leftarrow E[(B(x) - C)^+]$ / $E[B(x)]$;
end
return ob ;

7.5.4 Examples of Overbooking Limit Calculation

In this subsection, we give a brief example on the calculation of the overbooking limit for the above policies. We consider the price of a single time slot in a resource is fixed, with p = G$100 and $C = 50$ for simplicity. However, the denied cost $cost_{ds}$ and the show-rate q are varied from 125 to 175 and from 0.60 to 0.95, respectively. In this example, the money is represented in Grid dollar (G$).

$$\begin{aligned} ENR &= pE[B(ob)] - cost_{ds} * E[(B(ob) - C)^+] \\ &= p * ob * q - cost_{ds} * \sum_{k=C+1}^{ob} \binom{ob}{k} q^k (1-q)^{ob-k} \end{aligned} \tag{7.9}$$

Table 7.3 and 7.4 shows the overbooking limit (*ob*), expected net revenue (ENR in G$), and service level (SL) for each show rate (q), according to the probability-based and risk-based policies, respectively. Note that the ENR and SL in both tables are calculated using Equations 7.9 and 7.8 respectively.

For the probability-based policy, as q decreases, *ob* increases accordingly, as shown in Table 7.3. However, this policy does not take into a consideration the denied cost in its overbooking calculation. Thus, at each q, as $cost_{ds}$ increases, the ENR decreases.

On the other hand, the risk-based policy adaptively selects *ob* with a consideration of both the show rate and the denied cost, as shown in Table 7.4. In this example, the risk-based policy calculates more overbooking limit than the probability-based one, since the demand are forecasted to be greater than the cancellation rate (i.e. 1 - q). However, as $cost_{ds}$ increases at each q, *ob* decreases to prevent the ENR from reducing any further. As a result, the risk-based policy generates more net revenue than the probability-based

Table 7.3: Calculating the overbooking limit by using a Probability-based policy.

q	ob	Expected Net Revenue (G$)			Service Level
		$cost_{ds} = 125$	$cost_{ds} = 150$	$cost_{ds} = 175$	
0.60	83	4,770.5	4,728.6	4,686.7	0.0337
0.65	76	4,769.1	4,734.9	4,700.8	0.0277
0.70	71	4,796.7	4,762.1	4,727.4	0.0279
0.75	66	4,805.6	4,776.7	4,747.8	0.0233
0.80	62	4,828.4	4,802.1	4,775.8	0.0212
0.85	58	4,836.5	4,817.8	4,799.1	0.0152
0.90	55	4,870.7	4,854.9	4,839.0	0.0128
0.95	52	4,898.9	4,890.7	4,882.4	0.0067

Table 7.4: Calculating the overbooking limit by using a Risk-based policy.

q	$cost_{ds} = 125$			$cost_{ds} = 150$			$cost_{ds} = 175$		
	ob	ENR	SL	ob	ENR	SL	ob	ENR	SL
0.60	90	4,836.9	0.0834	87	4,750.4	0.0600	85	4,689.9	0.0459
0.65	83	4,846.7	0.0813	80	4,766.8	0.0555	78	4,711.1	0.0405
0.70	76	4,858.8	0.0693	74	4,784.2	0.0509	73	4,729.6	0.0425
0.75	71	4,870.4	0.0683	69	4,802.4	0.0480	68	4,753.2	0.0389
0.80	66	4,884.2	0.0600	64	4,824.3	0.0385	63	4,782.2	0.0292
0.85	62	4,898.4	0.0564	60	4,847.9	0.0330	59	4,811.5	0.0232
0.90	58	4,916.7	0.0465	57	4,873.1	0.0334	56	4,846.4	0.0219
0.95	54	4,941.4	0.0294	53	4,912.3	0.0162	53	4,891.9	0.0162

one for the same show rate and denied cost.

Table 7.5 shows the ob and ENR for a given service level, i.e. from 0.01 (1%) to 0.0001 (0.01%). Note that the ENR is also calculated by using Equation 7.9. From Table 7.5, it can be concluded that as SL decreases, ob and ENR become smaller for the same q. Although this policy produces the lowest net revenue of all the overbooking policies, it

Table 7.5: Calculating the overbooking limit by using a Service-level policy (with $cost_{ds} = 150$).

q	SL = 0.01		SL = 0.001		SL = 0.0001	
	ob	ENR	ob	ENR	ob	ENR
0.60	77	4,555.3	70	4,194.9	66	3,959.4
0.65	71	4,563.3	66	4,283.7	62	4,029.5
0.70	67	4,628.8	62	4,334.6	59	4,129.4
0.75	63	4,667.1	59	4,418.9	56	4,199.6
0.80	60	4,731.7	56	4,475.3	54	4,319.5
0.85	57	4,779.0	54	4,584.0	52	4,419.6
0.90	54	4,813.7	52	4,675.1	50	4,500.0
0.95	52	4,890.7	50	4,750.0	50	4,750.0

has the lowest number of denied reservations. For example, for the same $cost_{ds}$ of 150 in the service-level policy, the SL of the risk-based policy varies from 0.0162 ($q = 0.95$) to 0.0600 ($q = 0.60$), as shown in Table 7.4. This means that about 162 – 600 out of 10,000 reservations are denied by a resource provider when using the risk-based policy. For the probability-based policy, the SL varies from 0.0067 ($q = 0.95$) to 0.0337 ($q = 0.60$) regardless of any denied costs, as shown in Table 7.3. Thus, the resource provider needs to deny 67 – 337 out of 10,000 reservations when using this policy.

Overall, from this example, the probability-based policy can be used to generate an extra net income when the show rate is high (e.g. $q \geq 0.90$) and the denied cost is low (e.g. $cost_{ds} = 125$). Moreover, it has the simplest formula for calculating the overbooking limit. In contrast, the risk-based policy can be applied to produce more revenue, when the demand exceeds the cancellation rate and the denied cost increases over time. Finally, the service-level policy can be adopted to reduce the number of denied reservations due to overbooking in the long run.

7.5.5 Capacity Allocation with Overbooking

Algorithm 14: Capacity Allocation with Overbooking

1 $ob \leftarrow \mathit{OverbookingLimit}(q,\ maxCN)$;
2 $C^{+} \leftarrow \max(maxCN,\ ob)$;
3 $y_1 \leftarrow C^{+} - \mathit{BookingLimit}(C^{+},\ p_1,\ p_3,\ F_1)$;
4 $y_2 \leftarrow C^{+} - \mathit{BookingLimit}(C^{+},\ p_2,\ p_3,\ F_2)$;
5 $b_3 \leftarrow \max(0,\ C^{+} - y_1 - y_2)$;
6 $b_2 \leftarrow b_3 + \mathit{BookingLimit}(C^{+} - b_3,\ p_1,\ p_2,\ F_1)$;
7 $b_1 \leftarrow C^{+}$;

Algorithm 14 shows how an existing capacity allocation problem, as discussed in Section 7.4.3, can be extended to support overbooking. Initially, the overbooking limit needs to be calculated according to one of the models we have previously discussed (line 1). Then, a virtual capacity C^{+} can be found (line 2), where $C^{+} \geq maxCN$. Then, we calculate the protection levels of higher-class users, before the booking limit of class 3 users (line 3–5). Finally, we determine the booking limit of class 2 and 1 users (line 6–7). Note that line 3–7 are similar to the Algorithm 11, where we use C^{+} instead of $maxCN$ for the resource's maximum capacity.

7.6 Reservation Pricing, Penalty Fee and Denied Cost

Apart from overbooking and capacity allocation, the next important point in RM is to determine the pricing of each reservation. Moreover, if a cancellation or no-show occurs, a penalty fee needs to be introduced to discourage users from misusing AR, and to cover some operational cost associated with managing reservations. Finally, the denied cost due to overbooking also needs to be addressed.

7.6.1 Pricing of Reservations

As mentioned previously, we differentiate jobs based on whether they are using reservations or not. For non-reserved jobs, we calculate the running cost as

$$cost = dur * numCN * bcost \tag{7.10}$$

where *dur* denotes the job runtime, *numCN* denotes the number of compute nodes, and *bcost* is the base cost of running a job at one time unit. Intuitively, the cost for jobs that use AR will incur higher due to the privilege of having guaranteed resources at a future time. Hence, the running cost for reserved jobs is charged based on the number of reserved slots in the data structure or GarQ:

$$cost_{AR} = numSlot * numCN * bcost_{AR} \tag{7.11}$$

$$bcost_{AR} = \tau * bcost * \delta \tag{7.12}$$

where $numSlot$ is the total number of reserved slots, $bcost_{AR}$ is the cost of running the AR job at one time slot, and τ is a constant factor ($\tau \geq 1$) to differentiate reservation prices. With this equation, the RMS can simply modify the τ value, as necessary. Note that δ is a fixed time interval used by GarQ (as mentioned in Chapter 5).

Table 7.6 shows an example of setting different τ of Equation 7.12, where τ_1, τ_2, and τ_3 denote τ for user class 1, 2 and 3, respectively. In this table, we set $\tau > 1$, such that $cost_{AR} > cost$ for one time slot or δ time unit. In addition, for simplicity, we specify that τ_1 and τ_2 are 50% and 25% more expensive than τ_3, respectively. Then, we set τ_3

Table 7.6: An example of variable pricing with different τ_1, τ_2, and τ_3 during the week.

Pricing Name	Day Period	Time Period	τ_1	τ_2	τ_3
Super Saver	Weekdays	12 am – 06 am	1.88	1.56	1.25
Peak	Weekdays	06 am – 06 pm	3.38	2.81	2.25
Off-Peak	Weekdays	06 pm – 12 am	2.63	2.19	1.75
Super Saver	Weekends	06 pm – 06 am	1.88	1.56	1.25
Off-Peak	Weekends	06 am – 06 pm	2.63	2.19	1.75

differently for various time periods.

We classify the time period of weekdays into *peak*, *off-peak* and *super saver*, as shown in Table 7.6. For weekends, we only have *off-peak* and *super saver*. The main purpose of having this classification or variable pricing is to increase the resource revenue and provide nodes to applications that are highly valued by the users. For example, the RMS can increase τ_3 during the *peak* period to shift the demands of budget conscious users to other periods. Thus, the nodes can be reserved for users with tight deadlines, since they are willing to pay more. Note that the time period classification is based on the daily arrival rate of jobs recorded by several parallel and Grid systems [49, 85]. For simplicity, we partition the time period with either a 6-hour or 12-hour block.

7.6.2 Penalty Fee for Cancellations and No-Shows

We use a simple penalty fee calculation, where the RMS charges the user with a penalty rate α_p times the price for each canceled or no-show reservation, where $0 \leq \alpha_p \leq 1$. $\alpha_p = 0$ means the reservation is fully refundable, and $\alpha_p = 1$ means it is not refundable. For example, if $\alpha_p = 0.10$ and the reservation price is G$100, then the penalty fee would be G$10 or 10% of the price. In multiple fare classes, we have $\alpha_{p1} < \alpha_{p2} < \alpha_{p3}$.

7.6.3 Denied or Compensation Cost

We use Equation 7.11 to determine $cost_{ds}$, i.e. the denied service or compensation cost for each reservation. The value of τ_{ds} depends on the agreement or policy set by the resource provider to a particular user class, with $\tau_{ds} > \tau$. Moreover, we present several strategies for addressing which excess reservations to deny at the starting time t_s, based on $cost_{ds}$ and user class level, namely Lottery, Denied Cost First (DCF), and Lower Class DCF (LC-DCF), as shown in Algorithm 15, 16, and 17 respectively.

Algorithm 15: Lottery drawing

Input: t_s **and** C

```
1 bookingList ← get_booking_list(t_s) ;
2 overbookedCN ← get_total_CN(bookingList) − C;
3 denyCN ← 0;                                   // total nodes to be denied
4 while denyCN < overbookedCN do
5     data ← get_booking(bookingList, LOTTERY);
6     calculate_denied_cost(data) ;
7     remove(data, bookingList) ;
8     denyCN ← denyCN + get_total_CN(data) ;
9 end
```

The simplest way to deny existing reservations is by conducting a lottery drawing, as depicted in Algorithm 15. Initially, a list of bookings, *bookingList*, that start at time t_s is withdrawn from the data structure (line 1). Since a booking may require more than one node, we also need to find out the number of overbooked CNs, *overbookedCN*, based on the current capacity C, and the total CNs required from *bookingList* (line 2). Then, the algorithm performs a lottery drawing on *bookingList* (line 5), with the unlucky booking is compensated and removed from the list and the data structure altogether (line 6–7). Next, the total CNs to be denied, *denyCN*, is incremented (line 8). Finally, this algorithm keeps ejecting more bookings from the list as long as $denyCN < overbookedPE$ (line 4–9).

Algorithm 16: Denied Cost First (DCF)

Input: t_s **and** C

```
1 bookingList ← get_booking_list(t_s) ;
2 overbookedCN ← get_total_CN(bookingList) − C;
3 denyCN ← 0;                                   // total nodes to be denied
4 sort(bookingList, GLOBAL_DENIED_COST) ;
5 while denyCN < overbookedCN do
6     data ← get_booking(bookingList, HEAD) ;
7     calculate_denied_cost(data) ;
8     remove(data, bookingList) ;
9     denyCN ← denyCN + get_total_CN(data) ;
10 end
```

In contrast, to minimize the total compensation cost on t_s, the Denied Cost First (DCF) strategy chooses which bookings to be denied based on $cost_{ds}$, as shown in Algorithm 16. Thus, DCF sorts *bookingList* based on the lowest $cost_{ds}$ globally, regardless of any class types (line 3). Afterwards, DCF removes this booking from the head of *bookingList* (line

6), since the list is sorted from lowest to highest $cost_{ds}$. The rest of the operations are similar to the Lottery strategy.

Lower Class Denied Cost First (LC-DCF), as shown in Algorithm 17, has a similar strategy as DCF. However, LC-DCF aims at protecting higher-class bookings from being denied in the first place. Hence, LC-DCF sorts *bookingList* based on $cost_{ds}$ for each class type (line 3). Similar to DCF, LC-DCF removes a booking from the head of *bookingList* (line 6), but this booking is from a lower-class user that has the lowest $cost_{ds}$. If there are no more bookings from a lower class, then LC-DCF continues removing bookings from a higher class. The rest of the operations are similar to the Lottery strategy.

Algorithm 17: Lower Class Denied Cost First (LC-DCF)

Input: t_s **and** C

```
1  bookingList ← get_booking_list(t_s) ;
2  overbookedCN ← get_total_CN(bookingList) − C;
3  denyCN ← 0;                                   // total nodes to be denied
4  sort(bookingList, CLASS_DENIED_COST) ;
5  while denyCN < overbookedCN do
6      data ← get_booking(bookingList, HEAD) ;
7      calculate_denied_cost(data) ;
8      remove(data, bookingList) ;
9      denyCN ← denyCN + get_total_CN(data) ;
10 end
```

7.7 Performance Evaluation

In this section, we evaluate the effectiveness of using Revenue Management on Grid systems. We model these systems based on EU DataGrid TestBed I [48]. The testbed topology is shown in Figure 7.4. The details of simulation parameters are discussed next.

7.7.1 Simulation Setup

Table 7.7 summarizes all the resource relevant information, where we divide the resources into four VOs, based on their location. In GridSim, a CPU rating of one node is modeled in the form of Million Instructions Per Second (MIPS) as devised by Standard Performance Evaluation Corporation (SPEC) [135]. The resource settings were obtained from the current characteristics of the real LHC testbed [82]. We took the data about these resources

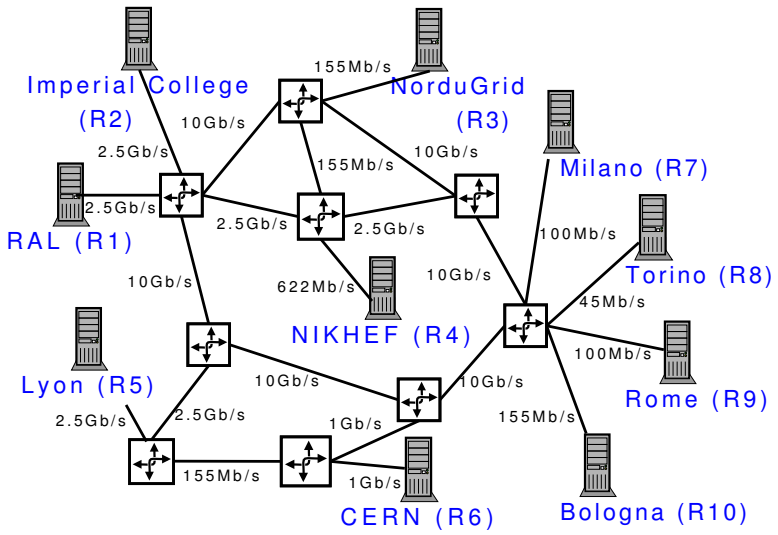


Figure 7.4: The simulated topology of EU DataGrid TestBed 1.

Table 7.7: Resource specifications and their jobs' inter-arrival rates (λ).

Resource Name (Location)	# Nodes	CPU Rating	VO	*bcost*	μ runtime	λ_{peak}	λ_{off}	λ_{saver}
RAL (UK)	41	49,000	1	0.49	3 hours	0.01670	0.00835	0.004175
Imperial College (UK)	52	62,000	1	0.62	3 hours	0.01670	0.00835	0.004175
NorduGrid (Norway)	17	20,000	2	0.20	3 hours	0.00835	0.004175	0.0020875
NIKHEF (Netherlands)	18	21,000	2	0.21	3 hours	0.00835	0.004175	0.0020875
Lyon (France)	12	14,000	3	0.14	3 hours	0.00835	0.004175	0.0020875
CERN (Switzerland)	59	70,000	3	0.70	3 hours	0.03340	0.00167	0.000835
Milano (Italy)	5	7,000	4	0.07	3 hours	0.00418	0.0020875	0.00104375
Torino (Italy)	2	3,000	4	0.03	3 hours	0.00167	0.000835	0.0004175
Rome (Italy)	5	6,000	4	0.06	3 hours	0.00418	0.00209	0.001045
Bologna (Italy)	67	80,000	4	0.80	3 hours	0.03340	0.0167	0.00835

and scaled the number of nodes of each resource by 10. This is because simulating original computing capacities is not possible due to limited physical memory in a computer, since many resources and jobs need to be created during the simulation.

In this experiment, we model the Grid systems based on the resources mentioned in Table 7.7. Thus, each resource has Revenue Management System (RMS), Resource Calendar and Resource Scheduler components, as depicted in Figure 7.1. For the Resource Calendar, GarQ is used with $\delta = 5$ minutes, and has a fixed interval length of 30 days. For the Resource Scheduler, the Easy Backfilling (BF) [98] policy is used. Then, we set all nodes in the resource to be homogeneous with the same CPU rating.

For calculating the pricing of reservations on each resource, we use Equation 7.11. Then, we apply the τ values shown in Table 7.6 into the equation. However, each resource

Table 7.8: μ CPU rating for Grid & VO level, and their jobs' inter-arrival rates (λ).

Level	μ Rating	μ runtime	λ_{peak}	λ_{off}	λ_{saver}
Grid	56,000	2 hours	0.13812	0.02290	0.01979
VO 1	56,000	5 hours	0.05087	0.02092	0.01913
VO 2	20,000	5 hours	0.05954	0.00537	0.00295
VO 3	60,000	5 hours	0.15901	0.00097	0.00046
VO 4	68,000	5 hours	0.07098	0.00672	0.00257

Table 7.9: Simulated users' characteristics.

Trace Level	User Category	Booking Period	Search Limit Time	λ_c	α_p	τ_{ds}
Grid	*Premium*	2 hours	2 hours	0.25	0%	5 τ_1
Resource	*Business*	4 hours	4 hours	0.45	10%	4 τ_2
VO	*Budget*	6 hours	24 hours	0.85	25%	3 τ_3

has different *bcost*, as shown in Table 7.7. We determine the *bcost* of each resource based on its CPU rating. Therefore, the higher the rating of a resource, the more costly it becomes. For example, RAL has a CPU rating of 49,000. Thus, its *bcost* is G$0.49, where we scale the rating by 100,000.

We model incoming job traffic at three levels: Grid (with all 10 resources), VO, and resource, by using a Poisson model with different lambdas for peak (λ_{peak}), off-peak (λ_{off}) and super saver (λ_{saver}) period, as depicted in Table 7.7 and 7.8. With these lambdas, we can set the peak period to be arriving more frequently than the off-peak period and so on. The lambdas for Grid and VO levels are taken from [85], where the authors used a 3-stage Markov Modulated Poisson Process (MMPP) model in their workload analysis.

For handling no-show of reservations, we use binomial distribution with the probability of no-shows (q_{ns}) sets to 0.05, 0.10 and 0.15 for peak, off-peak, and super saver periods respectively. For job runtime, we use an exponential distribution with different mean (μ) for each level. Since we are trying to simulate Bag-of-Tasks (BoT) applications, we set the number of reserved nodes to be 1 for all bookings.

We identify the Grid-level trace to be *Premium* users with a booking period of 2 hours and a search limit time of 2 hours, as depicted in Table 7.9. The search limit time is used for finding alternative time slots, if resources in the initial starting time are unavailable, as mentioned in Figure 6.3. Then, we choose each resource-level trace to be *Business* users, whereas each VO-level trace to be *Budget* users. All traces use exponential distribution

to calculate the number of cancellations. Table 7.9 also lists the job's canceled rate (λ_c), the penalty rate (α_p), and τ_{ds} for the denied service cost for each user class.

For the *Premium* users, they will choose a resource from the Grid based on the earliest job completion time, whereas for the *Budget* users, they will submit jobs to a resource within the VO based on the cheapest price. Since all resources have different CPU ratings, we scale each job duration in the trace according to the μ rating found in Table 7.8. However each *Business* user is designated to submit to a particular resource, so no scaling is required. For all users, if a booking for the current job can not be made due to unavailability of nodes, then we ignore this job and proceed to the next one. Overall, we simulate 15 traces in this evaluation for a period of 14 days.

The main objective of this experiment is to look at the impact of using RM in increasing the revenue of a resource. Therefore, we have two scenarios: in Scenario 1 (S1), we select $R1$ (RAL) and $R10$ (Bologna) to use a static pricing method with $\tau_s = \{1.9,\ 2.8\}$ (without RM), and $R2 - R9$ to have RM. Then in Scenario 2 (S2), all resources use RM by using variable τ according to Table 7.6.

Another objective is to examine the impact of the overbooking policies (Probability (Pr), Risk, and Service-Level (SL)), and the denied-booking strategies (Lottery, DCF, and LC-DCF) on the net revenue of a resource, where cancellations and no-shows are allowed. For the Service-Level (SL) policy, we set the $ds\_threshold$ to be 0.01 or 1%. We compare these scenarios with the same set of parameters.

7.7.2 Results

Table 7.10 shows the initial protection levels, y_1 and y_2 for the *Premium* and *Business* users respectively for S2. Based on this table, the *Budget* users are allocated to 25%, 50% and 75% of total capacity during peak, off-peak and super saver period respectively. Since y_1 and y_2 will be re-forecasted dynamically based on demand fluctuation, according to Algorithm 11, a resource provider is only required to give an initial estimation.

Table 7.11 shows the total revenue earned by each resource in both scenarios. $R1$ and $R10$ make a huge profit by using RM in S2, instead of using a static pricing of $\tau_s = 1.9$ in S1. This is because $R1$ and $R10$ protect some free nodes for the *Premium* and *Business* users' bookings at later times, since they pay at a higher rate of τ compared to $\tau_s = 1.9$

Table 7.10: Initial protection levels, y_1 and y_2.

Resource Name	Peak		Off-Peak		Super Saver	
	y_1	y_2	y_1	y_2	y_1	y_2
RAL (R1)	10	20	5	15	3	7
Imperial (R2)	12	27	6	20	2	11
NorduGrid (R3)	5	8	2	6	1	3
NIKHEF (R4)	5	8	2	6	1	3
Lyon (R5)	3	6	2	4	0	3
CERN (R6)	12	32	6	23	3	11
Milano (R7)	1	2	0	2	0	1
Torino (R8)	0	1	0	0	0	0
Rome (R9)	1	2	0	2	0	1
Bologna (R10)	15	35	8	25	4	12

Table 7.11: Total revenue for each resource, where $\tau_s = 1.9$ in S1 for RAL and Bologna.

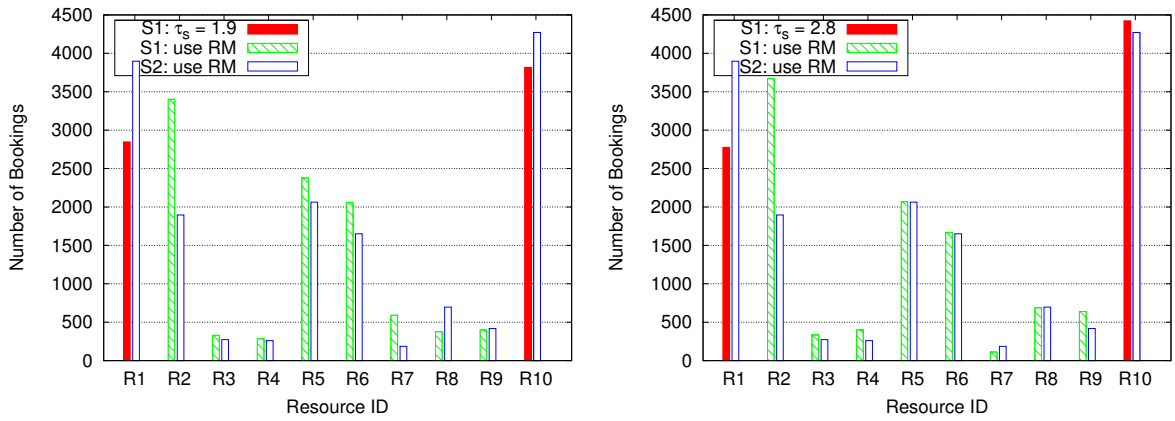
Resource	S1 (x1000)	S2 (x1000)	% gain / loss
RAL (R1)	G$ 834	G$ 31,523	3,678.70
Imperial (R2)	G$ 66,662	G$ 61,645	-7.53
NorduGrid (R3)	G$ 2,570	G$ 4,638	80.44
NIKHEF (R4)	G$ 4,928	G$ 5,171	4.94
Lyon (R5)	G$ 684	G$ 742	8.37
CERN (R6)	G$ 101,997	G$ 103,529	1.50
Milano (R7)	G$ 170	G$ 171	0.57
Torino (R8)	G$ 10	G$ 13	26.46
Rome (R9)	G$ 114	G$ 119	4.07
Bologna (R10)	G$ 2,051	G$ 147,279	7,079.71

and τ_3. However, RM also provide a limited number of available nodes with a cheaper price to the *Budget* users. According to Table 7.6, the average of τ_3 is 1.65 or at least 15% cheaper than $\tau_s = 1.9$. As a result, RM is beneficial to both time- and budget-conscious users, and resource providers.

As expected, when we increase τ_s in S1 from 1.9 to 2.8 (average of τ during peak period), $R1$ and $R10$ both gained an extra 42% and 39% respectively in the total revenue, as shown in Table 7.12. However, from Table 7.11 and 7.12, we can see that by adopting RM, $R1$ and $R10$ produce more profits in comparison to using a static pricing. Even with the increased of τ_s, the percentage gain for both $R1$ and $R10$ is still very large, i.e. more than 2,500%, as highlighted in Table 7.12. The main reason is that during peak period in S2, more nodes are available for the *Premium* users in $R1$ and $R10$, since they pay a higher τ compared to $\tau_s = 2.8$.

Table 7.12: Total revenue for each resource, where $\tau_s = 2.8$ in S1 for RAL and Bologna.

Resource	S1 (x1000)	S2 (x1000)	% gain / loss
RAL (R1)	G$ 1,188	G$ 31,523	2,553.44
Imperial (R2)	G$ 70,437	G$ 61,645	-12.48
NorduGrid (R3)	G$ 4,656	G$ 4,638	-0.38
NIKHEF (R4)	G$ 5,117	G$ 5,171	1.06
Lyon (R5)	G$ 854	G$ 742	-13.15
CERN (R6)	G$ 103,184	G$ 103,529	0.34
Milano (R7)	G$ 209	G$ 171	-18.13
Torino (R8)	G$ 15	G$ 13	-15.66
Rome (R9)	G$ 157	G$ 119	-24.45
Bologna (R10)	G$ 2,863	G$ 147,279	5,043.37



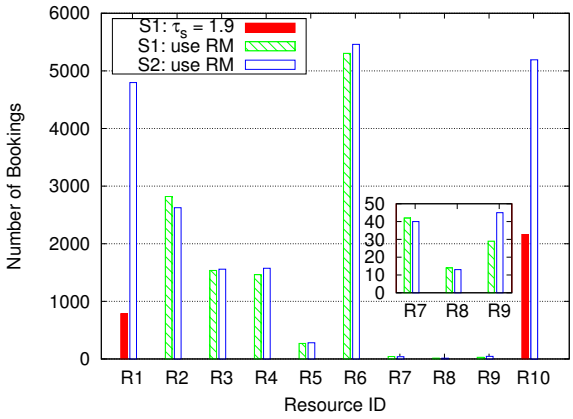
(a) Using $\tau_s = 1.9$ in Scenario 1 (S1).

(b) Using $\tau_s = 2.8$ in Scenario 1 (S1).

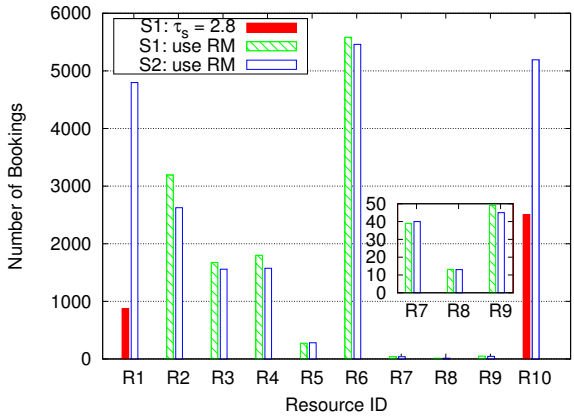
Figure 7.5: Total number of bookings for *Budget* users.

Figure 7.5–7.7 show the total number of bookings made by each resource for different user classes. When $R1$ uses a static pricing of τ_s in S1, *Budget* users within `VO_1` prefer to send their jobs to $R2$, due to a cheaper price in the VO during off-peak and super saver period, as depicted in Figure 7.5(a) and 7.5(b). However, when $R1$ adopts RM in S2, more bookings from these users are being made. The same trend can be observed for $R10$ in `VO_4` in Figure 7.5(a). For Figure 7.5(b), $R10$ in S2 sets a limited quota on the *Budget* users. Therefore, there is a slight decrease on the number of bookings for this user class.

Figure 7.6 and 7.7 show the bookings made by the *Business* and *Premium* users respectively. Due to the fact that no protection levels are imposed on $R1$ and $R10$ in S1, when they want to book closer to the reservation time, no available nodes can be found. As a consequence, the *Business* users have to cancel their bookings, and the *Premium* users have to use other resources in the Grid. This situation is called *dilution*, since $R1$

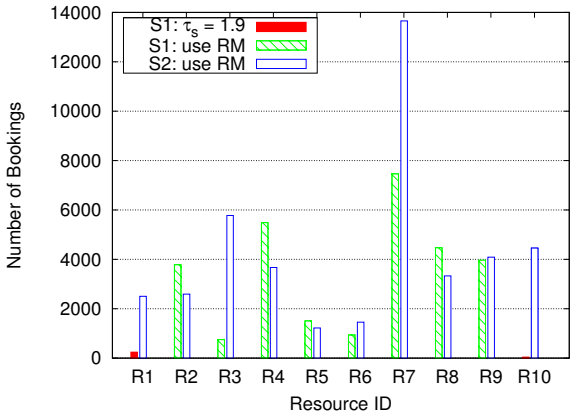


(a) Using $\tau_s = 1.9$ in Scenario 1 (S1).

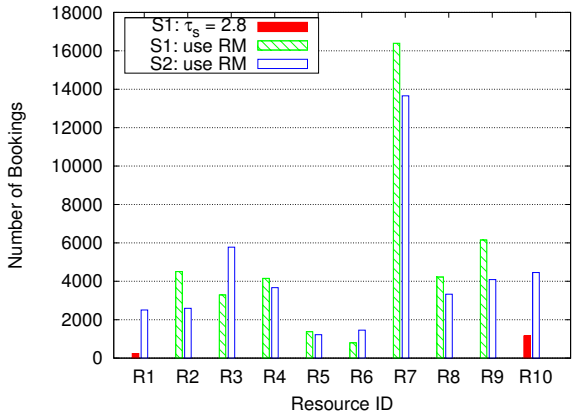


(b) Using $\tau_s = 2.8$ in Scenario 1 (S1).

Figure 7.6: Total number of bookings for *Business* users.



(a) Using $\tau_s = 1.9$ in Scenario 1 (S1).



(b) Using $\tau_s = 2.8$ in Scenario 1 (S1).

Figure 7.7: Total number of bookings for *Premium* users.

and $R10$ decrease the revenue they would have received from protecting additional nodes, y_1 and y_2, for these users.

When $R1$ and $R10$ utilizing RM in S2, the number of bookings are significantly grown for the *Business* and *Premium* users. As a result, $R1$ and $R10$ are experiencing a huge increase in the revenue, as shown in Table 7.11 and 7.12. However, the increased number of bookings have an effect in other resources, as depicted in Figure 7.5 and 7.7. This is because the *Budget* and *Premium* users can book to any available resources within the VO and Grid respectively. Among other resources, the impact was felt by $R2$ the hardest in Table 7.11, since $R2$ is located on the same VO as $R1$. For Table 7.12, $R7$–$R9$ lose the most revenue, since both them and $R10$ belong to `VO_4`.

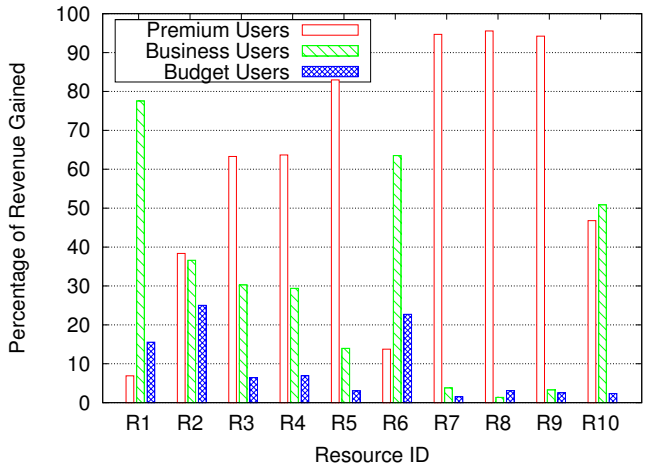


Figure 7.8: Percentage of income revenue in scenario 2 (S2 - all resources using RM).

Figure 7.8 shows the percentage of incoming revenue for each user class. For smaller and medium-sized resources, such as Torino ($R8$) and NorduGrid ($R3$), the *Premium* users contribute more than 60% of the total revenue. On the other hand, the *Business* users contribute more than 50% on large-sized resources, such as CERN ($R6$) and Bologna ($R10$). Hence, from this figure, both the *Business* and *Premium* users are a major source of revenue for a resource. Therefore, in a competitive market, a resource needs to differentiate itself among others to attract these users.

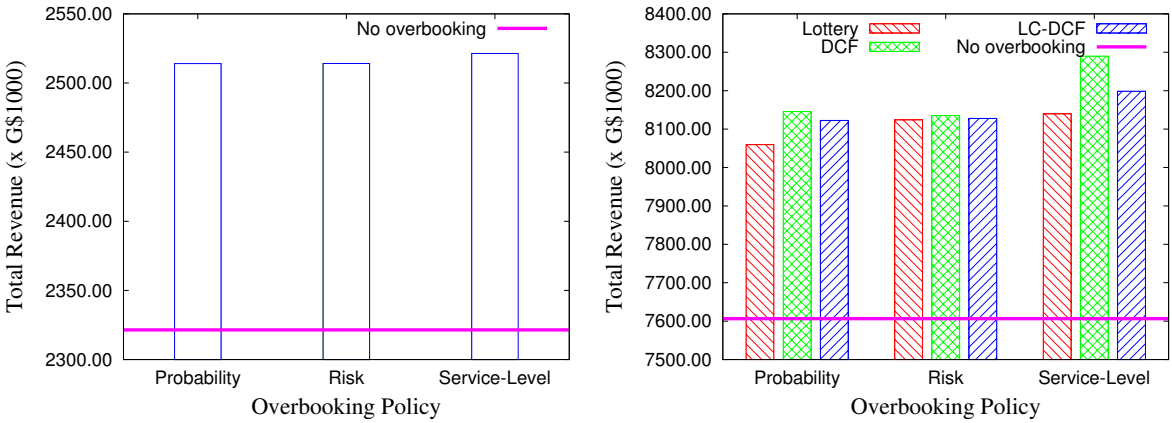
7.7.3 Results using Overbooking

Table 7.13 shows the negative effect of unanticipated cancellations and no-shows (CNS) on the net revenue of each resource. By allowing CNS and without any overbooking policies, all resources experienced a significant drop in revenue, i.e. by more than 87%. However, if we set RAL and Bologna to use overbooking policies instead, they both reported around 6–9% increase in net profits from their previous evaluation (without overbooking), as shown in Figure 7.9(a) and 7.9(b) respectively. Thus, this finding provides a financial incentive for other resources to overbook. Note that RAL has zero denied bookings for all the overbooking policies. Hence, in this section, we mainly discuss the impact of overbooking policies and denied-booking strategies in Bologna.

When looking at the performance of each overbooking policy in Figure 7.9(a) and 7.9(b), all policies produce about the same amount of net revenue. However, the main difference

Table 7.13: The impact of unanticipated cancellations and no-shows (CNS) on net revenue.

Resource Name	No CNS (x 1000)	Allow CNS (x 1000)	% loss
RAL	G$ 31,523	G$ 2,321.44	-92.64
Imperial	G$ 61,645	G$ 7,038.12	-88.58
Nordu	G$ 4,638	G$ 413.90	-91.08
NIKHEF	G$ 5,171	G$ 421.90	-91.84
Lyon	G$ 742	G$ 94.08	-87.32
CERN	G$ 103,529	G$ 10,400.91	-89.95
Milano	G$ 171	G$ 7.46	-95.63
Torino	G$ 13	G$ 0.58	-95.43
Rome	G$ 119	G$ 5.56	-95.31
Bologna	G$ 147,279	G$ 7,606.47	-94.84



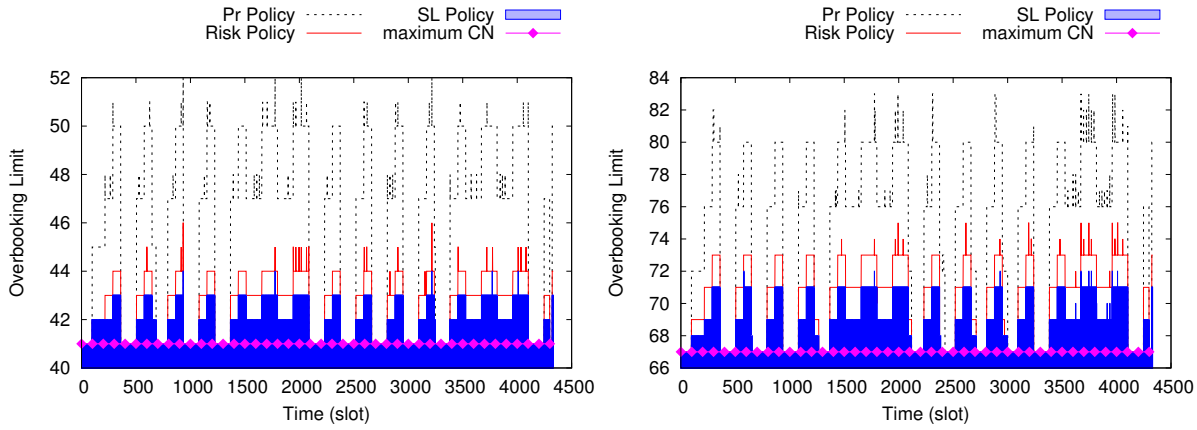
(a) RAL. (b) Bologna.

Figure 7.9: Total net revenue.

between them is the overbooking limit, *ob*, at each time slot in the data structure, as shown in Figure 7.10(a) and 7.10(b). Note that we omit figures using Lottery and LC-DCF in Bologna, since they are similar.

For RAL in Figure 7.10(a), the maximum *ob* percentage gain from $maxCN$ is 7%, 12% and 27% for SL, Risk and Pr policies respectively. For Bologna in Figure 7.10(b), it is 8%, 12% and 24% for SL, Risk and Pr policies respectively. Thus, in both cases, the SL policy is the most conservative of all, since it estimates the lowest *ob*. This is consistent with the calculation that we performed in Section 7.5.4. However, with $cost_{ds}$ can be up to five times more expensive than $cost_{AR}$, the Risk policy sets a lower limit than the Pr policy in both Figure 7.10(a) and 7.10(b).

In this evaluation, we found that a lower *ob* leads to a smaller the total number of denied bookings and compensation cost, as shown in Figure 7.11 and 7.12 for Bologna



(a) RAL.

(b) Bologna (using DCF).

Figure 7.10: Overbooking Limit.

respectively. In both figures, on average, the Risk and SL policies are 49% and 74% lower than the Pr policy respectively.

Apart from estimating *ob*, another important issue is selecting which excess bookings to deny. In terms of total net revenue, the denied-booking strategies (Lottery, DCF, and LC-DCF) in Bologna produced a similar income, i.e. within 0.1–2% of each other, as shown in Figure 7.9(b). On average, DCF gives the highest total amount of income, followed by LC-DCF and then Lottery.

Surprisingly, the Lottery strategy has the lowest total denied bookings compared to DCF and LC-DCF in the Pr and Risk policies, as shown in Figure 7.11. The Lottery strategy is 4% and 35% lower than DCF in the Pr and Risk policies respectively. Moreover, it is 12% and 40% lower than LC-DCF in the Pr and Risk policies respectively. For the SL policy, the Lottery strategy is 2% higher than DCF, but 27% lower than LC-DCF, as depicted in Figure 7.11. As a result, the Lottery strategy works best in reducing total denied bookings. Moreover, it is the simplest and easiest to implement.

However, each denied booking has a different value in terms of the job duration time, user class level, and more importantly $cost_{ds}$. Thus, due to its randomness, the Lottery strategy pays the most amount of money to denied users, by up to 16%, 10% and 65% in the Pr, Risk and SL policies respectively, compared to DCF and LC-DCF, as depicted in Figure 7.12. Hence, from the compensation cost's point of view, the Lottery strategy is the least desirable.

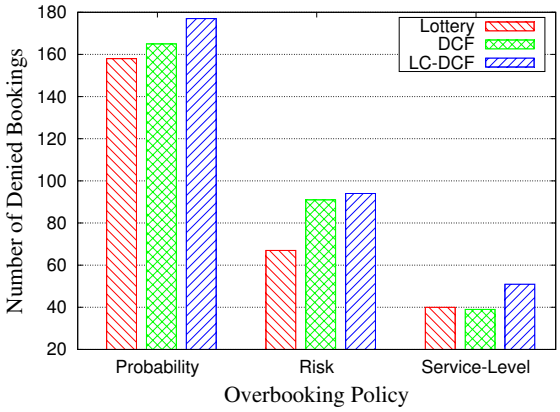


Figure 7.11: Denied bookings for Bologna (lower number is better).

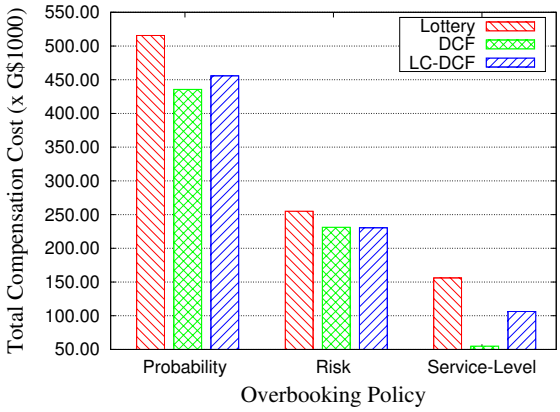


Figure 7.12: Total denied cost for Bologna (lower number is better).

Measuring DCF against LC-DCF, DCF has the lowest number of denied bookings by 7%, 3% and 30% in the Pr, Risk and SL policies respectively, as highlighted in Figure 7.11. Hence, in terms of total $cost_{ds}$, DCF is about 5% and 94% lower than LC-DCF in the Pr and SL policies respectively, as indicated in Figure 7.12. For the Risk policy, both DCF and LC-DCF have a similar cost, less than 0.5% of each other. This is because the Risk policy calculates the overbooking limit carefully based on the denied cost. When combining with the SL policy, the total net revenue with DCF is the highest of all, as shown in Figure 7.9(b). Overall, from these findings, DCF seems to be a better choice than LC-DCF.

From Figure 7.8, we found out that *Premium* and *Business* users contribute more than 60% on smaller and medium-sized resources (e.g. Torino and NorduGrid), and 50% on large-sized resources (e.g. CERN and Bologna) respectively. The main disadvantage of DCF is that this strategy does not take into consideration which user class level each booking belongs to. In contrast, LC-DCF removes bookings from lower-class users first, based on their $cost_{ds}$. As a result, LC-DCF has the lowest number of denied *Premium* users, as shown in Table 7.14. Therefore, to minimize the negative effects from high-paying users who have been denied access, the combination of SL and LC-DCF policies is a better solution in the long run.

Table 7.14: Total denied bookings for the Service-Level policy.

	Premium Users	*Business* Users	*Budget* Users
Lottery	14	23	1
DCF	15	33	2
LC-DCF	6	38	7

7.8 Related Work

Several studies have been done to improve handling and scheduling of reservations in Grid systems with some degree of flexibilities using different techniques [76, 116, 124]. However, [116, 124] provide a simple pricing model to determine the usage cost of each reservation. This may not be sufficient, as resources need to adopt a more complex method to increase their incentives or profits in a competitive market.

Numerous economic models for resource management have been proposed in the literature. These include: commodity market models [20, 150], tendering or contract-net models [81, 138], auction models [114, 152, 107], bid-based proportional resource sharing models [80], and cooperative bartering models [35]. From these models, RM is more suited to the commodity market one, where it complements the commodity's pricing. So far, RM techniques have been widely adopted in various industries, such as airlines, hotels, and car rentals [92].

In a study done by Smith et al. [131] on American Airlines, 50% of the bookings were resulted in cancellations or no-shows. Moreover, the report found that 15% of the flight seats would be unused, if bookings were only limited to the capacity of a plane. Therefore, overbooking models were introduced to address the problem in unanticipated cancellations and no-shows, by several researchers in the airlines industry [29, 37, 118, 145, 127]. The overbooking policies were also studied and applied to various industry, such as hotel [14, 42, 64], cruise [147], air cargo [75], health [78] and car rentals [27, 58]. Similarly, in this chapter, we adopt these overbooking policies in the context of scheduling jobs by means of reservations in a Grid resource. Moreover, we propose several strategies for determining which excess reservations to deny, based on compensation cost and user class level.

In networks, overbooking is used to optimize throughput [95] and to address the issue of burst contentions in optical burst switched networks from a new domain [160]. Similarly, in

Grids, Urgaonkar et al. [148] suggested overbooking as a way to increase resource utilization in shared hosting platforms, by specifying an overbooking tolerance on each component of an application running on one compute node. However, none of these works aim at maximizing revenue by charging the users with different prices, and calculating an ideal overbooking limit.

7.9 Summary

This chapter presents a novel approach of using Revenue Management (RM) to determine the pricing of reservations in Grid systems in order to increase resource revenue. The main objective of RM is to maximize profits by providing the right price for every product to different customers, and periodically update the prices in response to market demands. Therefore, a resource provider can apply RM techniques to *shift demands* requested by budget conscious users to off-peak periods as an example. As a result, more resources are available for users with tight deadlines in peak periods that are willing to pay more.

We evaluate the effectiveness of RM and show that by segmenting users, charging them with different pricing schemes, and protecting resources for those who are willing to pay more, will result in an increase of total revenue for that resource, by at least 25-fold.

In addition, this thesis suggests the concept of *overbooking* in order to protect the resource against unexpected cancellations and no-shows of reservations. By overbooking, the resource accepts more reservations than the maximum capacity. Thus, it can be effectively used to minimize the loss of revenue. This chapter adopts several static overbooking policies, such as Probability (Pr), Risk, and Service-Level (SL). In addition, this thesis introduces several novel strategies to select which excess bookings to deny, based on compensation cost and user class level, namely Lottery, Denied Cost First (DCF), and Lower Class DCF.

The result shows that the Pr policy suffers from excessive denied bookings and compensation cost ($cost_{ds}$), since it calculates the overbooking limit (ob) based only on user demands at that particular time. The Risk policy manages to balance the show rate and $cost_{ds}$ well. However, it tends to set a more conservative ob, when the compensation cost is much higher than the weighted average price of all class users. Finally, the SL policy

defines a specified level or fraction of denied users. This approach has the advantage of having the lowest denied bookings and $cost_{ds}$ compared to other policies.

With regards to the denied-booking strategies, the result shows that DCF to be the best as it has both the lowest $cost_{ds}$ compared to Lottery and LC-DCF, and the highest net revenue when associated with the SL policy. However, to prevent high-paying users from submitting their jobs to other resources due to overbooking, the combination of the SL and LC-DCF policies is the better option. Overall, the result indicates that by overbooking reservations, the resource gains of an extra 6-9% in the total net revenue is achievable. Thus, this finding shows a financial incentive for resources to overbook.

Chapter 8

Conclusion and Future Directions

Grid technologies represent a significant achievement towards the aggregation of networked resources for solving large-scale data-intensive or compute-intensive applications [52]. This thesis proposes the use of advance reservation to ensure the specified resources are available for applications when required. In addition, this thesis recommends the use of Revenue Management to determine the pricing of reservations, increase resource revenue, and regulate supply and demand. These studies are carried out through modeling and simulation on GridSim, a discrete-event Grid simulation tool, since different scenarios need to be evaluated and repeated. In this chapter, we highlight the thesis contributions and present possible future directions.

8.1 Conclusion

This thesis describes the development of GridSim, which allows modeling and simulation of various properties, such as differentiated level of network Quality of Service (QoS), data Grid and resource discovery in a virtual organization (VO). This thesis also introduces the work done on GridSim to support advance reservation. These features of GridSim provide essential building blocks for simulating various Grid scenarios. Thus, GridSim offers researchers the functionality and flexibility of simulating Grids for various types of studies, such as service-oriented computing [39], Grid meta-scheduling [3], workflow scheduling [113], VO-oriented resource allocation [44], and security solutions [101].

In addition, several improvements to the existing GridSim design were performed in order to make it more flexible and extensible. As a result, new features can be added and incorporated easily into GridSim for the performance evaluation on topics addressed in this thesis. These topics include modeling and scheduling of task graphs with advance reservation and interweaving, using an elastic reservation approach on Grid systems, and adapting Revenue Management techniques to determine the pricing of reservations.

This thesis presents a novel approach to schedule task graphs by using advance reservation in a homogeneous environment, such as cluster computing. In addition, to improve the resource utilization, this thesis proposes an advance reservation (AR) scheduler by interweaving one or more task graphs within the same reservation block, and backfilling with other independent jobs (if applicable).

By interweaving a set of task graphs, the AR scheduler manages to reduce the overall reservation duration time up to 26.31% on 4 compute nodes (CNs). In addition, when there are many small independent jobs, the AR scheduler is able to fill these jobs into the reservation blocks. As a result, the AR scheduler improves the utility of the system substantially on a cluster with 16 and 32 nodes compared to the First Come First Serve (FCFS) and EASY backfilling algorithms.

This thesis provides a case for an elastic reservation model, where users can *self-select* or choose the best option in reserving their jobs, according to their QoS needs, such as deadline and budget. In this model, each Grid system has a Reservation System (RS) and a Resource Calendar (ResCal). The RS is responsible for handling reservation queries and requests. When the RS receives a reservation query or request, it searches for availability. More specifically, the RS communicates with the ResCal for this request. Therefore, the primary role of the ResCal is to store and update information about resource availability as time progresses.

A well-designed data structure provides the flexibility and easiness in implementing various algorithms. This thesis suggests an array-based data structure for administering reservations efficiently in the ResCal. The new data structure is called Grid advanced reservation Queue (GarQ).

GarQ is a *time-slotted* structure, where each slot contains rv, the number of already reserved nodes, and a linked list for storing reservations that start at this time. Thus, it

partitions the duration or length of a reservation into slots based on a fixed time interval δ. If the duration spans multiple slots, rv on each of them is updated accordingly. GarQ has the following advantages: (i) a fast $O(1)$ access to a particular slot; (ii) able to reuse these slots for the next time interval, assuming that the length of a reservation is less than 30 days; and (iii) built only once in the beginning.

This thesis adapts an On-line Strip Packing (OSP) algorithm for the RS. The OSP algorithm considers the duration and number of required compute nodes as soft constraints for a given reservation query. Thus, it aims to find a solution or alternative offers within the given time interval for users to choose themselves. In addition, the OSP algorithm aims to reduce fragmentations or idle time gaps caused by having reservations in the system.

Having a degree of flexibility in the reservation requests allows an improvement in the resource utilization. Results show that the elastic model improves the resource utilization by 4.39% on average compared to the rigid model. In addition, the elastic model reduces the number of rejections by 54.88% on average compared to the rigid model. The results also show that by allowing users to select an alternative offer if no solutions are found, the OSP algorithm reduces the total number of rejection by around 13.5% – 63.6%. Note that the rigid model treats all the request parameters as hard constraints. Therefore, if no solution is found, then the rigid model will reject such requests.

The challenging issue of adopting AR in existing Grid systems is its impact in increasing the waiting times of local jobs in the queue. Smith et al. [132] showed that providing AR capabilities increases waiting times of applications in the queue by up to 37%. As expected, results show that the rigid model has a minimal impact on the average waiting time, as it did not accept too many reservations. However, the elastic model performs better as the reservation requests become more flexible. The results show that the elastic model improves its performance by 22% on average. The elastic model performs better than the rigid model for requests with a book-ahead time of 10 hours.

This thesis proposes the use of Revenue Management (RM) to determine the pricing of reservations in Grid systems in order to increase resource revenue. The main objective of RM is to maximize profits by providing the right price for every product to different customers, and periodically update the prices in response to market demands. Therefore, a resource provider can apply RM techniques to *shift demands* requested by budget conscious

users to off-peak periods as an example. As a result, more resources are available for users with tight deadlines in peak periods who are willing to pay more for the privilege. With the adaption of RM, the functionalities of the Reservation System are integrated into the Revenue Management System (RMS).

By segmenting customers, charging them with different pricing schemes, and protecting resources for those who are willing to pay more, the result shows an increase of total revenue by at least 25-fold. In addition, using RM techniques ensure that resources are allocated to applications that are highly valued by the users.

However, in reality, users may cancel their reservations before starting time or by not submitting at all (*no-show*), due to reasons such as resource or network failures on the other end. Thus, during a period of high demands for example, the resource provider has no choice but to reject bookings from potential users, who are committing to use the resource and willing to pay for a higher price. As a result, the resource provider is faced with a prospect of loss of income and lower system utilization.

This thesis suggests the concept of *overbooking* in order to protect the resource against unexpected cancellations and no-shows of reservations. By overbooking, the resource accepts more reservations than the maximum capacity. Thus, it can be effectively used to minimize the loss of revenue. In addition, this thesis introduces several novel strategies to select which excess bookings to deny, based on compensation cost and user class level, namely Lottery, Denied Cost First (DCF), and Lower Class DCF. The result shows that the DCF produced the lowest total compensation cost compared to other strategies. The result also indicates that by overbooking reservations, the resource gains of an extra 6-9% in the total net revenue is achievable. Thus, this finding shows a financial incentive for resources to overbook.

8.2 Future Directions

This thesis suggests several future directions to further enhance advance reservation and revenue-based resource management for Grid systems. The future directions are related to the three key functionalities of Grids, i.e. job scheduling, resource management and data management.

8.2.1 Incorporating Resource Failure Model

The Resource Scheduler presented in this thesis assume that all the compute nodes are available for execution. However, in reality, some of these nodes may not be available at some point in the future due to maintenance or upgrade (e.g. software, hardware and security). Thus, the Resource Scheduler needs to consider a case where several nodes fail during execution.

The addition of a resource failure model to the job scheduling problem will present another challenge to the Resource Scheduler and RMS. The Resource Scheduler needs to interact with the RMS and the Resource Calendar to find suitable solutions. Such solutions can be migrating the affected jobs to other available nodes either located internally or externally, postponing these jobs to later times, or providing them with some compensation costs. However, these decisions needed to be chosen carefully as they may reduce the overall resource revenue and disrupt other reservations and existing jobs in the queues. As such, incorporating the resource failure model provides an interesting and exciting research problem.

8.2.2 Addressing Complex Reservation Scenarios

The work presented in this thesis uses real workload and synthetic traces. These traces provide a duration time for each job, based on information recorded on production parallel systems [49] or the exponential distribution. However, in reality, users may under- or over-estimate their jobs' duration time.

In case of over-estimation, this will introduce problems, such as finding available nodes for other reservations or incurring longer waiting time for non-reserved jobs. Thus, the Resource Scheduler and RMS need to consider this issue. One possible solution is to allow users to specify what to expect in case their reservations finish early or late. This is a similar approach undertaken by the Dynamic Soft Real-Time (DSRT) Scheduling System [77], as mentioned in Section 2.2.2. Another solution is to partition the resource or system, i.e. to allocate certain amount of nodes (not all) for AR. Thus, the remaining nodes can act as a buffer against over-estimation and overbooking.

With regards to the Revenue Management's overbooking and tactic, a scenario where

cancellations and no-shows are dependent of the total bookings need to be considered. In addition, a scenario for handling group bookings and cancellations need to be analyzed. These scenarios will have a significant effect on the calculation of booking limit for each user class and the total net revenue.

8.2.3 Integrating Various Types of Resources

As mentioned previously, common resources that can be reserved are compute nodes (CNs), storage elements (SEs), network bandwidth or a combination of any of those. However, this thesis is mainly focusing on reserving compute nodes. Therefore, allowing users to reserve a combination of resource types is highly desirable, since various applications, especially in the area of data Grid, can be modeled and studied.

This work leads to another interesting research problem, as it involves coordination and negotiation of multiple resources shared by different organizations. In the case of reserving network bandwidth, a Network Manager is needed to focus on network management issues, such as establishing a guaranteed end-to-end path, and handling traffic congestion. In addition, implementing the Multi Protocol Label Switching (MPLS) architecture [117] into GridSim may also be required. In the case of data Grid applications, a Replica Manager is needed to address various data management issues, such as deletion and replication of data sets. Thus, to reserve a combination of resource types, the RMS needs to collaborate with the Network Manager and Replica Manager.

This work also brings an issue in determining the overall reservation price, as each resource type may have its own price model. A feasible solution is to establish a multilateral pricing agreement between the resource providers. The agreement may include the base cost of using the resources, discount rate, penalty rate, and compensation cost.

8.2.4 Implementing Resource Management on a Real Grid Testbed

The study about resource management in this thesis is carried out through modeling and simulation on GridSim, since different scenarios need to be evaluated and repeated. The next step is to turn this study into a reality, i.e. by implementing the RMS, the Resource Calendar, and the Resource Scheduler as a prototype on a real Grid testbed. The prototype can be built on top of open-source software, such as Maui Scheduler [91]

or Sun Grid Engine [123], as discussed in Section 2.2.1 and 2.2.4, respectively. Thus, this work leads to a number of challenging software engineering issues, such as system reliability or persistence, handling simultaneous transactions, and testing and debugging the prototype source code.

References

[1] D. Abramson, J. Giddy, and L. Kotler. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? In *Proceedings of the 14th International Parallel and Distributed Processing Symposium(IPDPS'00)*, Cancun, Mexico, May 1–5, 2000.

[2] T. L. Adam, K. M. Chandy, and J. Dickson. A comparison of list scheduling for parallel processing systems. *Communications of the ACM*, 17:685–690, 1974.

[3] V. Agarwal, G. Dasgupta, K. Dasgupta, A. Purohit, and B. Viswanathan. DECO: Data replication and Execution CO-scheduling for Utility Grids. In *Proceedings of the 4th International Conference on Service Oriented Computing*, Chicago, USA, Dec. 4–7, 2006.

[4] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(8):872–892, Sep. 1998.

[5] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[6] International Virtual Observatory Alliance. http://www.ivoa.net, 2008.

[7] Avaki EII - Enterprise Data Integration Sofware. http://www.sybase.com/products/allproductsa-z/avakieii, 2008.

[8] R. Bagrodia, R. Meyer, M. Takai, Y. an Chen, X. Zeng, J. Martin, and H. Y. Song. Parsec: A Parallel Simulation Environment for Complex Systems. *Computer*, 31(10):77–85, Oct. 1998.

[9] W. H. Bell, D. G. Cameron, L. Capozza, A. P. Millar, K. Stockinger, and F. Zini. Simulation of Dynamic Grid Replication Strategies in OptorSim. In *Proceedings of the 3rd International Workshop on Grid Computing (Grid'02)*, Baltimore, USA, Nov. 2002.

[10] P. P. Belobaba. Application of a probabilistic decision model to airline seat inventory control. *Operations Research*, 37(2):183–197, 1989.

[11] J. O. Berkey and P. Y. Wang. Two-dimensional finite bin-packing algorithms. *Operational Research Society*, 38(5):423–429, 1987.

[12] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS project: Software support for high-level grid application development. *High Performance Computing Applications*, 15(4):327–344, 2001.

[13] Biogrid Project. http://www.biogrid.jp/e/project/index.html, 2008.

[14] G. R. Bitran and S. M. Gilbert. Managing Hotel Reservations with Uncertain Arrivals. *Operations Research*, 44(1):35–49, 1996.

[15] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC 2475: An Architecture for Differentiated Service. http://www.ietf.org/rfc/rfc2475.txt, Dec. 1998.

[16] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, editors. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. W3C, 2006. http://www.w3c.org/TR/xml.

[17] A. Brodnik and A. Nilsson. A static data structure for discrete advance bandwidth reservations on the internet. In *Proceedings of Swedish National Computer Networking Workshop (SNCNW'03)*, Stockholm, Sweden, Sep. 2003.

[18] R. Brown. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[19] L.-O. Burchard. Analysis of data structures for admission control of advance reservation requests. *IEEE Transactions on Knowledge and Data Engineering*, 17(3), 2005.

[20] 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 & Exhibition on High Performance Computing in Asia-Pacific Region (HPC Asia'00)*, Beijing, China, May 2000.

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

[22] R. Buyya and M. Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Management and Scheduling for Grid Computing. *Concurrency and Computation: Practice and Experience (CCPE)*, 14:13–15, 2002.

[23] R. Buyya and S. Venugopal. The Gridbus Toolkit for Service Oriented Grid and Utility Computing: An Overview and Status Report. In *Proceedings of the 1st International Workshop on Grid Economics and Business Models (GECON'04)*, Seoul, Korea, April 23, 2004.

[24] A. Caminero, A. Sulistio, B. Caminero, C. Carrion, and R. Buyya. Extending GridSim with an Architecture for Failure Detection. In *Proceedings of the 13th International Conference on Parallel and Distributed Systems (ICPADS'07)*, Hsinchu, Taiwan, Dec. 5–7, 2007.

[25] M. Cannataro and D. Talia. The Knowledge Grid. *Communications of the ACM*, 46(1):89–93, 2003.

[26] M. Caramia, S. Giordani, and A. Iovanella. Grid scheduling by on-line rectangle packing. *Network*, 44(2):106–119, 2004.

[27] W. Carroll and R. Grimes. Evolutionary change in Product Management Experiences in the Rental Car Industry. *Interfaces*, 25:84–104, 1996.

[28] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experimentations. In *Proceedings of the 10th International Conference on Computer Modeling and Simulation (UKSim'08)*, Cambridge, UK, April 2008.

[29] R. E. Chatwin. Multiperiod Airline Overbooking with a Single Fare Class. *Operations Research*, 46(6):805–819, 1999.

[30] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Network and Computer Applications*, 23:187–200, 2001.

[31] L. Childers, T. Disz, R. Olson, M. E. Papka, R. Stevens, and T. Udeshi. Access Grid: Immersive Group-to-Group Collaborative Visualization. In *Proceedings of the 4th International Immersive Projection Technology Workshop*, Ames, USA, June 19–20, 2000.

[32] H.-H. Chu and K. Nahrstedt. CPU Service Classes for Multimedia Applications. In *Proceedings of the 6th International Conference on Multimedia Computing and Systems (ICMCS'99)*, Florence, Italy, June 7-11, 1999.

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

[34] E. G. Coffman, editor. *Computer and Job-Shop Scheduling Theory*. Wiley, 1976.

[35] 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'02)*, Vienna, Austria, July 2–5, 2002.

[36] L. B. Costa, L. Feitosa, E. Araujo, G. Mendes, R. Coelho, W. Cirne, and D. Fireman. MyGrid: A complete solution for running bag-of-tasks applications. In *Proceedings of the Simposio Brasileiro de Redes de. Computadores (SBRC'04)*, Gramado, Brazil, May 2004.

[37] J. Coughlan. Airline Overbooking in the Multi-Class Case. *Operational Research Society*, 50(11):1098–1103, 1999.

[38] M. D. de Assuncao and R. Buyya. An Evaluation of Communication Demand of Auction Protocols in Grid Environments. In *Proceedings of the 3rd International Workshop on Grid Economics and Business (GECON'06)*, Singapore, May 16, 2006.

[39] M. D. de Assuncao, W. Streitberger, T. Eymann, and R. Buyya. Enabling the Simulation of Service-Oriented Computing and Provisioning Policies for Autonomic Utility Grids. In *Proceedings of the 4th International Workshop on Grid Economics and Business (GECON'07)*, Rennes, France, Aug. 28, 2007.

[40] W. F. de la Vega and V. Zissimopoulos. An approximation scheme for strip packing of rectangles with bounded dimensions. *Discrete Appl. Math.*, 82(1-3):93–101, 1998.

[41] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflow onto the grid. In *Across Grids Conference 2004*, Nicosia, Cyprus, 2004.

[42] F. DeKay, B. Yates, and R. Toh. Non-performance Penalties in the Hotel Industry. *Hospitality Management*, 23(3):273–286, 2004.

[43] C. L. Dumitrescu and I. Foster. GangSim: A Simulator for Grid Scheduling Studies. In *Proceedings of the 5th International Symposium on Cluster Computing and the Grid (CCGrid'05)*, Cardiff, UK, May 9–12, 2005.

[44] E. Elmroth and P. Gardfjall. Design and Evaluation of a Decentralized System for Grid-wide Fairshare Scheduling. In *Proceedings of the 1st International Conference on e-Science and Grid Computing (e-Science'05)*, Melbourne, Australia, Dec. 5–8, 2005.

[45] K. B. Erickson, R. E. Ladner, and A. Lamarca. Optimizing static calendar queues. *ACM Trans. on Modeling and Computer Simulation*, 10(3):179–214, 2000.

[46] Energy Sciences Network. http://www.es.net, 2008.

[47] EU Data Mining Grid. http://www.datamininggrid.org, 2008.

[48] The European DataGrid Project. http://eu-datagrid.web.cern.ch/eu-datagrid, 2008.

[49] D. Feitelson. Parallel Workloads Archive. http://www.cs.huji.ac.il/labs/parallel/workload, 2008.

[50] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing (NPC'06))*, Tokyo, Japan, Oct. 2–4, 2006.

[51] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Supercomputer Applications*, 11(2):115–128, 1997.

[52] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, San Francisco, USA, 1999.

[53] I. Foster, C. Kesselman, C. Lee, R. 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'99)*, London, UK, June 1–4, 1999.

[54] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *High Performance Computing Applications*, 15(3):200–222, 2001.

[55] I. Foster, A. Roy, and V. Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *Proceedings of the 8th International Workshop on Quality of Service (IWQoS'00)*, Pittsburgh, USA, June 5–7, 2000.

[56] G-lambda. http://www.g-lambda.net, 2008.

[57] The EU Project – GEANT2. http://www.geant2.net, 2008.

[58] M. K. Geraghty and E. Johnson. Revenue Management Saves National Car Rental. *Interfaces*, 27:107–127, 1997.

[59] H. Gibbins, K. Nadiminti, B. Beeson, R. Chhabra, B. Smith, and R. Buyya. The Australian BioGrid Portal: Empowering the Molecular Docking Research Community. In *Proceedings of the 3rd APAC Conference and Exhibition on Advanced Computing, Grid Applications and eResearch (APAC'05)*, Gold Coast, Australia, Sep. 26–30, 2005.

[60] S. Graupner, J. Pruyne, and S. Singhal. Making the Utility Data Center a Power Station for the Enterprise Grid. Technical Report HPL–2003–53, HP Labs, Palo Alto, USA, 2003.

[61] J. Gray and L. Lamport. Consensus on Transaction Commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[62] C. Guok, D. W. Robertson, M. R. Thompson, J. Lee, B. Tierney, and W. Johnston. Intra and Interdomain Circuit Provisioning Using the OSCARS Reservation System. In *Proceedings of the 3rd International Conference on Broadband Communications, Networks, and Systems (BROADNETS'06)*, San Jose, CA, USA, Oct. 1–5, 2006.

[63] The Grid Workloads Archive. http://gwa.ewi.tudelft.nl, 2008.

[64] G. C. Hadjinicola and C. Panayi. The Overbooking Problem in Hotels with Multiple Tour Operators. *Operations and Production Management*, 17(9):874–885, 1997.

[65] U. Hoenig and W. Schiffmann. A comprehensive test bench for the evaluation of scheduling heuristics. In *Proceedings of the 16th International Conference on Parallel and Distributed Computing and Systems (PDCS'04)*, Cambridge, USA, Nov. 9–11, 2004.

[66] U. Hoenig and W. Schiffmann. Improving the Efficiency of Functional Parallelism by Means of Hyper-Scheduling. Workshop on Compile and Runtime Techniques for Parallel Computing. In *Proceedings of the 35th International Conference on Parallel Processing (ICPP'06)*, Ohio, USA, Aug. 14–18, 2006.

[67] The Hybrid Optical and Packet Infrastructure Project. http://networks.internet2.edu/hopi/, 2008.

[68] W. Hoschek, F. J. Jaén-Martínez, A. Samar, H. Stockinger, and K. Stockinger. Data management in an international data grid project. In *Proceedings of the 1st International Workshop on Grid Computing (Grid'00)*, Bangalore, India, Dec. 17, 2000.

[69] C. Hu, J. Huai, and T. Wo. Flexible resource reservation using slack time for service grid. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS'06)*, Minneapolis, USA, July 12–15, 2006.

[70] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[71] S.-Y. Hwang and B. Riddle. BRUW: Bandwidth Reservation for User Work. In *Proceedings of the TERENA Networking Conference 2005*, Poznan, Poland, June 6–9, 2005.

[72] International Air Transport Association (IATA). *Airline Passengers Call for More Self-Service*. Number 43 in Press Releases. IATA Corporate Communications, Nov. 20, 2007.

[73] Internet2 Network. http://www.internet2.edu/network/, 2008.

[74] J. C. Jacob, D. S. Katz, T. Prince, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, G. Singh, and M.-H. Su. The Montage Architecture for Grid-enabled Science Processing of Large, Distributed Datasets. In *Proceedings of the Earth Science Technology Conference (ESTC'04)*, June 22–24, 2004.

[75] R. G. Kasilingam. An Economic Model for Air Cargo Overbooking under Stochastic Capacity. *Computers and Industrial Engineering*, 32(1):221–226, 1997.

[76] N. R. Kaushik, S. M. Figueira, and S. A. Chiappari. Flexible time-windows for advance reservation scheduling. In *Proceedings of the 14th International Symposium on Modeling, Analysis, and Simulation (MASCOTS'06)*, California, USA, Sep. 11–13, 2006.

[77] K. Kim. Extended DSRT System. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (USA), Aug. 2000.

[78] S. Kim and R. E. Giachetti. A Stochastic Mathematical Appointment Overbooking Model for Healthcare Providers to Improve Profits. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 36(6):1211–1219, 2006.

[79] Y.-K. Kwok and I. Ahmad. Link Contention-constrained Scheduling and Mapping of Tasks and Messages to a Network of Heterogeneous Processors. *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, 3(2):113–124, 2000.

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

[81] S. Lalis and A. Karipidis. JaWS: An Open Market-Based Framework for Distributed Computing over the Internet. In *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (Grid'00)*, Bangalore, India, Dec. 17, 2000.

[82] LCG Computing Fabric Area. http://lcg-computing-fabric.web.cern.ch, 2008.

[83] C. C. Lee and D. T. Lee. A simple on-line bin-packing algorithm. *J. ACM*, 32(3):562–572, 1985.

[84] H. Li, D. Groep, and L. Walters. Workload characteristics of a multi-cluster supercomputer. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 176–193. Springer-Verlag, 2004. Lect. Notes Comput. Sci. vol. 3277.

[85] H. Li and M. Muskulus. Analysis and modeling of job arrivals in a production grid. *SIGMETRICS Performance Evaluation Review*, 34(4):59–70, 2007.

[86] J. MacLaren, editor. *Advance Reservations: State of the Art (draft)*. GWD-I, Global Grid Forum (GGF), June 2003. http://www.ggf.org.

[87] J. MacLaren. Co-allocation of Compute and Network resources using HARC. In *Proceedings of Lighting the Blue Touchpaper for UK e-Science: Closing Conference of the ESLEA Project*, Edinburgh, UK, March 26–28, 2007.

[88] J. MacLaren. HARC: The Highly-Available Resource Co-allocator. In *Proceedings of the International Conference on Grid computing, high-performAnce and Distributed Applications (GADA'07)*, Vilamoura, Algarve, Portugal, Nov. 29–30, 2007.

[89] E. Mannie. RFC 3945: Generalized Multi-Protocol Label Switching (MPLS) Architecture. http://www.ietf.org/rfc/rfc3945.txt, Oct. 2004.

[90] S. Martello, M. Monaci, and D. Vigo. An exact approach to the strip-packing problem. *INFORMS J. on Computing*, 15(3):310–319, 2003.

[91] Maui Cluster Scheduler. http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php, 2008.

[92] J. I. McGill and G. J. V. Ryzin. Revenue Management: Research Overview and Prospects. *Transportation Science*, 33(2):233–256, 1999.

[93] S. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlington. Workflow enactment in ICENI. *UK e-Science All Hands Meeting*, pages 894–900, Sep. 2004.

[94] E. Medernach. Workload analysis of a cluster in a grid environment. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 36–61. Springer-Verlag, June 2005.

[95] J. Milbrandt, M. Menth, and J. Junker. Experience-based Admission Control with Type-Specific Overbooking. In *Proceedings of the 6th International Workshop on IP Operations and Management (IPOM'06)*, Dublin, Ireland, Oct. 23–25, 2006.

[96] M. J. Mineter, C. H. Jarvis, and S. Dowers. From stand-alone programs towards grid-aware services and components: a case study in agricultural modelling with interpolated climate data. *Environmental Modelling and Software*, 18(4):379–391, April 2003.

[97] Moab workload manager. http://www.clusterresources.com/pages/products/moab-cluster-suite/workload-manager.php, 2008.

[98] 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, 2001.

[99] K. Nahrstedt, H. Chu, and S. Narayan. QoS-aware Resource Management for Distributed Multimedia Applications. *Journal on High-Speed Networking*, 7(3–4):227–255, 1998.

[100] S. Naiksatam and S. Figueira. Elastic Reservations for Efficient Bandwidth Utilization in LambdaGrids. *Future Generation Computer Systems*, 23(1):1–22, Jan. 2007.

[101] S. Naqvi and M. Riguidel. Grid Security Services Simulator (G3S) – A Simulation Tool for the Design and Analysis of Grid Security Solutions. In *Proceedings of the 1st International Conference on e-Science and Grid Computing (e-Science'05)*, Melbourne, Australia, Dec. 5–8, 2005.

[102] B. Nitzberg, J. M. Schopf, and J. P. Jones. PBS Pro: Grid Computing and Scheduling Attributes. In *Grid Resource Management: State of the Art and Future Trends*, pages 183–190. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[103] Ns-2 network simulator. http://www.isi.edu/nsnam/ns, 2008.

[104] D. of Transportation. Air Travel Consumer Report. In *Office of Aviation Enforcement and Proceedings (OAEP)*, Washington, DC, USA, April 2007.

[105] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[106] T. O'Reilly. What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html, Sep. 30, 2005.

[107] P. Padala, C. Harrison, N. Pelfort, E. Jansen, M. 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 (ISPDS'03)*, Ljubljana, Slovenia, Oct. 13–14, 2003.

[108] A. Patil, B. Belter, A. Polyrakis, M. Przybylski, T. Rodwell, and M. Grammatikou. GEANT2 Advance Multi-domain Provisioning System. In *Proceedings of the TERENA Networking Conference 2006*, Catania, Italy, May 15–18, 2006.

[109] PBS Pro. http://www.pbsgridworks.com/, 2008.

[110] G. F. Pfister. *In Search of Clusters*. Prentice Hall, 2nd edition, 1998.

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

[112] M. Priestley. *Practical Object-Oriented Design with UML*. McGraw-Hill Higher Education, 2nd edition, Dec. 2003.

[113] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi. Scheduling Data-intensive Workflows onto Storage-constrained Distributed Resources. In *Proceedings of the 7th International Symposium on Cluster Computing and the Grid (CCGrid'07)*, Rio de Janeiro, Brazil, May 14–17, 2007.

[114] 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)*, Charleston, USA, 1998.

[115] W. T. Rhee. Optimal bin packing with items of random sizes. *Math. Oper. Res.*, 13(1):140–151, 1988.

[116] T. Roeblitz, F. Schintke, and A. Reinefeld. Resource reservations with fuzzy requests. *Concurrency and Computation: Practice & Experience (CCPE)*, 18(13):1681–1703, 2006.

[117] E. Rosen, A. Viswanathan, and R. Callon. RFC 3031: Multiprotocol Label Switching Architecture. http://www.ietf.org/rfc/rfc3031.txt, Jan. 2001.

[118] M. Rothstein. Airline Overbooking: The State of the Art. *Transport Economics and Policy*, 5:96–99, 1971.

[119] A. Roy and V. Sander. *Advance Reservation API.* Scheduling Working Group, Global Grid Forum (GGF), GFD–E.5 edition, May 23, 2002.

[120] O. Schelén, A. Nilsson, J. Norrgård, and S. Pink. Performance of QoS agents for provisioning network resources. In *Proceedings of the 7th IFIP International Workshop on QoS (IWQoS'99)*, London, UK, June 1999.

[121] A. Schill, F. Breiter, and S. Kuhn. Design and evaluation of an advance reservation protocol on top of rsvp. In *Proceedings of the IFIP TC6/WG6.2 4th International Conference on Broadband Communications (BC'98)*, Stuttgart, Germany, April 1998.

[122] K. Seymour, A. YarKhan, S. Agrawal, and J. Dongarra. NetSolve: Grid Enabling Scientific Computing Environments. In L. Grandinetti, editor, *Grid Computing and New Frontiers of High Performance Processing*, volume 14 of *Advances in Parallel Computing*, pages 33–51, Netherlands, 2005. Elsevier.

[123] Sun Grid Engine. http://gridengine.sunsource.net, 2008.

[124] M. Siddiqui, A. Villazon, and T. Fahringer. Grid capacity planning with negotiation-based advance reservation for optimized QoS. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC'06)*, Florida, USA, Nov. 2006.

[125] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):75–87, 1993.

[126] C. Simatos. Making SimJava Count. Master's thesis, The University of Edinburgh (UK), Sep. 12, 2002.

[127] J. L. Simon. An Almost Practical Solution to Airline Overbooking. *Transport Economics and Policy*, 2:201–202, 1968.

[128] Simscript: a simulation language for building large-scale, complex simulation models. http://www.simscript.org, 2008.

[129] O. Sinnen and L. Sousa. On Task Scheduling Accuracy: Evaluation Methodology and Results. *Journal of Supercomputing*, 27(2):177–194, 2004.

[130] G. Sipos and P. Kacsuk. Multi-Grid, Multi-User Workflows in the P-GRADE Portal. *Journal of Grid Computing*, 3(3–4):221–238, Sep. 2005.

[131] B. C. Smith, J. F. Leimkuhler, and R. M. Darrow. Yield Management at American Airlines. *Interfaces*, 22:8–31, 1992.

[132] W. Smith, I. Foster, and V. Taylor. Scheduling with advanced reservations. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'00)*, Cancun, Mexico, May 1–5, 2000.

[133] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien. The MicroGrid: A Scientific Tool for Modeling Computational Grids. *Scientific Programming*, 8(3):127–141, 2000.

[134] GridSim Website. http://www.gridbus.org/gridsim, 2008.

[135] Standard Performance Evaluation Corporation. http://www.spec.org, 2008.

[136] Standard Task Graph Set. http://www.kasahara.elec.waseda.ac.jp/schedule/, 2008.

[137] H. Stockinger. *Database Replication in World-wide Distributed Data Grids*. PhD thesis, Fakultät für Wirtschaftswissenschaften und Informatik, Universität Wien, 2001.

[138] 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)*, Austin, USA, Sep. 28–30, 1994.

[139] A. Sulistio, U. Cibej, S. Venugopal, B. Robic, and R. Buyya. A Toolkit for Modelling and Simulating Data Grids: An Extension to GridSim. *Concurrency and Computation: Practice and Experience (CCPE)*, 20(13):1591–1609, Sep. 2008.

[140] A. Sulistio, G. Poduval, R. Buyya, and C.-K. Tham. On Incorporating Differentiated Levels of Network Service into GridSim. *Future Generation Computer Systems*, 23(4):606–615, May 2007.

[141] A. Takefusa, M. Hayashi, N. Nagatsu, H. Nakada, T. Kudoh, T. Miyamoto, T. Otani, H. Tanaka, M. Suzuki, Y. Sameshima, W. Imajuku, M. Jinno, Y. Takigawa, S. Okamoto, Y. Tanaka, and S. Sekiguchi. G-lambda: Coordination of a Grid Scheduler and Lambda Path Service over GMPLS. *Future Generation Computer Systems*, 22(8):868–875, Oct. 2006.

[142] K. T. Talluri and G. J. V. Ryzin. *The Theory and Practice of Revenue Management*. Springer Science + Bussiness Media, Inc., 2004.

[143] M. Tang, B.-S. Lee, C.-K. Yeo, and X. Tang. Dynamic Replication Algorithms for the Multi-tier Data Grid. *Future Generation Computer Systems*, 21(5):775–790, May 2005.

[144] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 17(2–4):323–356, 2005.

[145] H. R. Thompson. Statistical Problems in Airline Reservation Control. *Operations Research Quaterly*, 12:167–185, 1961.

[146] B. Tierney, J. Lee, L. T. Chen, H. Herzog, G. Hoo, G. Jin, and W. E. Johnston. Distributed Parallel Data Storage Systems: A Scalable Approach to High Speed Image Servers. In *Proceedings of the 2nd ACM International Conference on Multimedia*, San Francisco, USA, Oct. 15–20, 1994.

[147] R. S. Toh, M. J. Rivers, and T. W. Ling. Room Occupancies: Cruise Lines Out-do the Hotels. *Hospitality Management*, 24(1):121–135, 2005.

[148] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design & Implementation (OSDI'02)*, Dec. 9–11, 2002.

[149] S. Venugopal, R. Buyya, and K. Ramamohanarao. A Taxonomy of Data Grids for Distributed Data Sharing, Management and Processing. *ACM Computing Surveys*, 38(1):1–53, March 2006.

[150] S. Venugopal, R. Buyya, and L. Winton. A grid service broker for scheduling e-science applications on global data grids: Research articles. *Concurrency and Computation: Practice and Experience (CCPE)*, 18(6):685–699, 2006.

[151] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience (CCPE)*, 13(8–9):643–662, 2001.

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

[153] T. Wang and J. Chen. Bandwidth tree – a data structure for routing in networks with advanced reservations. In *Proceedings of the 21st International Performance, Computing, and Communications Conference (PCC'02)*, pages 37–44, Phoenix, USA, 2002.

[154] M. Y. Wu and D. D. Gayski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, 1990.

[155] Q. Xiong, C. Wu, J. Xing, L. Wu, and H. Zhang. A linked-list data structure for advance reservation admission control. In *Proceedings of the 3rd International Conference on Networking and Mobile Computing (ICCNMC'05)*, Zhangjiajie, China, Aug. 2–4 2005.

[156] C. S. Yeo, R. Buyya, M. D. de Assuncao, J. Yu, A. Sulistio, S. Venugopal, and M. Placek. Utility Computing and Global Grids. In H. Bidgoli, editor, *The Handbook of Computer Networks*, volume III Part 1, New York, USA, 2007. John Wiley & Sons.

[157] J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 3(3–4):171–200, 2005.

[158] L. Yuan, C.-K. Tham, and A. L. Ananda. A probing approach for effective distributed resource reservation. In *Proceedings of the 2nd International Workshop on Quality of Service in Multiservice IP Networks (QoS-IP'03)*, Milan, Italy, Feb. 2003.

[159] L. Zhang, S. Berson, S. Herzog, and S. Jamin. RFC 2205: Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. http://www.ietf.org/rfc/rfc2205.txt, Sep. 1997.

[160] Y.-X. Zhao and C.-J. Chen. A Redundant Overbooking Reservation Algorithm for OBS/OPS Networks. *Computer Networks*, 51(13):3919–3934, 2007.