Cost-Effective Resource Management for Distributed Computing

Mohd Amril Nurman Mohd Nazir

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of the

University of London.

Department of Computer Science

University College London

23rd August 2011

I, Mohd Amril Nurman Mohd Nazir, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

Current distributed computing and resource management infrastructures (e.g., Cluster and Grid) suffer from a wide variety of problems related to resource management, which include scalability bottleneck, resource allocation delay, limited quality-of-service (QoS) support, and lack of cost-aware and service level agreement (SLA) mechanisms.

This thesis addresses these issues by presenting a cost-effective resource management solution which introduces the possibility of managing geographically distributed resources in resource units that are under the control of a Virtual Authority (VA). A VA is a collection of resources controlled, but not necessarily owned, by a group of users or an authority representing a group of users. It leverages the fact that different resources in disparate locations will have varying usage levels. By creating smaller divisions of resources called VAs, users would be given the opportunity to choose between a variety of cost models, and each VA could rent resources from resource providers when necessary, or could potentially rent out its own resources when underloaded. The resource management is simplified since the user and owner of a resource recognize only the VA because all permissions and charges are associated directly with the VA. The VA is controlled by a 'rental' policy which is supported by a pool of resources that the system may rent from external resource providers. As far as scheduling is concerned, the VA is independent from competitors and can instead concentrate on managing its own resources. As a result, the VA offers scalable resource management with minimal infrastructure and operating costs.

We demonstrate the feasibility of the VA through both a practical implementation of the prototype system and an illustration of its quantitative advantages through the use of extensive simulations. First, the VA concept is demonstrated through a practical implementation of the prototype system. Further, we perform a cost-benefit analysis of current distributed resource infrastructures to demonstrate the potential cost benefit of such a VA system. We then propose a costing model for evaluating the cost effectiveness of the VA approach by using an economic approach that captures revenues generated from applications and expenses incurred from renting resources. Based on our costing methodology, we present rental policies that can potentially offer effective mechanisms for running distributed and parallel applications without a heavy upfront investment and without the cost of maintaining idle resources. By using real workload trace data, we test the effectiveness of our proposed rental approaches.

Finally, we propose an extension to the VA framework that promotes long-term negotiations and rentals based on service level agreements or long-term contracts. Based on the extended framework, we present new SLA-aware policies and evaluate them using real workload traces to demonstrate their effectiveness in improving rental decisions.

Acknowledgements

It has been a great pleasure working with the Department of Computer Science staff and students at University College London (UCL) and with the countless people who have provided me with aid and support during the five years it has taken to complete this thesis.

First of all, I want to express my gratitude to my Supervisor, Dr. Søren-Aksel Sørensen, for teaching me through his vision, for his consideration of every one of the countless emails I continue to send him (no matter how misguided they are), and for the pragmatism and soundness of his advice over the last five years.

It has also been a great honour working with brilliant young researchers on the 7th and 8th floor. The discussions with them have always enhanced my research and clarified my questions. In particular, I would like to thank my colleague Dr. Hao Liu for our exciting and detailed discussions on our PhD work, which sometimes went on for hours without us realising. I believe our collaboration has been both fun and fruitful.

I would like to thank Professor Dror Feitelson from the Hebrew University of Jerusalem, and Dr. Moe Jette from the Lawrence Livermore National Lab, United States for providing all the workload trace data available online on the Parallel Workloads Archive. Without them, this thesis may well have taken a few more years to complete.

I would like to thank Professor Philip Treleaven for introducing me to UCL. I am very grateful to Philip Morgan for his time and effort in proof reading my thesis. I would also like to acknowledge the Ministry of Science, Technology & Innovation, Malaysia (MOSTI) and the Malaysian Institute of Microelectronic Systems Berhad (MIMOS) for their financial support.

I am indebted to Professor David Rosenblum for his important administrative support and arranging my viva examination while I was away from the UK. Without his help, my thesis would not have been submitted on time.

I would like to express my gratitude to the many anonymous reviewers who have reviewed my papers and have given me so many constructive comments and suggestions. Their valuable feedback has led to the successful publication of some of the ideas that are central to my PhD thesis.

Finally, I would like to thank my parents for their support and encouragement over the years. These five years of hardship would have never been possible without constant support from them. Thank you very much for believing in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The way in which computers are used has changed significantly in recent years. The emergence of small electronic devices with excellent display capabilities and graphical user interfaces has completed the natural evolution towards an interactive information society that was started by the desktop computer. Users now expect such personal devices to perform tasks that are well beyond their processing capabilities and the idea of a portable, handheld 'supercomputer' no longer seems a science fiction concept. At the same time the supercomputer concept has changed significantly. By connecting hundreds of readily available workstations as cluster systems, it is now possible to generate virtual computers with processing capabilities that rival those of the high-cost, dedicated supercomputer. Cluster systems are widely used by various organizations to accommodate the ever-increasing demand for computer resources from users and applications. Clusters are normally managed by Distributed Resource Managers (DRMs) (Henderson, 1995; Gentzsch, 2001a; Xu, 2001; Jackson, Snell, and Clement, 2001; Frey et al., 2002) to provide users with simple access to computational resources to run resource hungry applications.

The advent of high-speed networks has enabled the integration of computational resources which are geographically distributed and administered at different domains. Such integration is known as 'Grid computing' and enables many independent computers to be connected together as if they were one large parallel machine, or virtual supercomputer, to be used for solving large-scale problems in science and engineering (Foster and Kesselman, 1997).

In a more recent development, 'Cloud computing' (Sullivan, 2009) offers a solution that allows resource machines to be customized and dynamically configured using virtualization technology so that resources can be delivered on demand. Such an advance is important since the trend in distributed computing is to build and aggregate computing power composed from many geographically distributed resources.

1.1 Problem Statement

Computational approaches to problem solving have proven their worth in almost every field of human endeavour. Computers are used for modelling and simulating complex scientific and engineering problems, diagnosing medical conditions, controlling industrial equipment, forecasting the weather, managing stock portfolios, and many other purposes. Computer simulations are used in a wide variety of practical contexts, such as analysis of air pollutant dispersion using atmospheric dipersion modelling, behaviour of structures (such as buildings and industrial parts) under stress and other conditions, reservoir simulation for petroleum engineering to model subsurface reservoirs, urban simulation models that simulate dynamic patterns of urban development and responses to urban land use and transportation policies, traffic engineering to plan or redesign parts of a street network from single junctions to a national highway network, for transportation system planning, design and operations, and many other purposes (Foster, Kesselman, and Tuecke, 2001). These simulation applications can be broadly categorized as either High Throughput Computing (HTC) or High Performance Computing (HPC) applications (Calleja et al., 2005). HTC applications are characterized by a large number of independent small-sized calculations, with moderate resource requirements, that can be distributed on commodity machines such as clusters. Execution can typically be spread out over long periods of time; days, weeks or even months. On the other hand, HPC applications are compute intensive and usually require the co-located use of large amounts of resources for a shorter period of time. Such jobs will tend to require tighter coupling, and parallel applications fall into this category.

More recently, large-scale scientific computing is playing an ever-increasing role in critical decision-making and dynamic, event-driven systems. There is a growing number of problem domains where key decisions must be made quickly with the aid of large-scale computation. In these domains, “urgent computing” is essential, and late results are useless (Beckman et al., 2006). For example, a computation to predict coastline flooding or avalanche danger must guide evacuation while there is still time. Therefore, in such a time-critical scenario, it is crucial to satisfy application-specific QoS requirements such as job/task deadlines. Furthermore, although not as urgent, on-demand computing is often required to take advantage of a scientific opportunity, for example, to process data and steer activities during an experiment or observation of an unpredictable natural event (Cencerrado, Senar, and Cortés, 2009). Without immediate access to large computational resources, the steering opportunities may not be possible.

Supercomputers have traditionally been used to provide this type of immense computational processing capability, but due to the low cost of PC computers or workstations, it is now more cost effective to build ‘supercomputers’ by connecting hundreds of cheap workstations to generate high processing capability. One well-known example is the Beowulf (Ridge et al., 1997), which was a supercomputer-like system created from a collection of a number of desktop PCs connected by a high-speed network. This concept is known as Cluster computing (Feitelson and Rudolph, 1995; Moreira et al., 1997; Weissman and Zhao, 1998) and the computer cluster has become a viable choice for solving large-scale computational problems since it is able to offer an equally high performance with a lower price compared with traditional super-computing systems. The emergence of clusters was initiated by a number of academic projects, such as Berkeley NOW (Culler, 1997), and HPVM (Chien et al., 1997) that proved the advantage of clusters over traditional HPC platforms.

In recent years, Grid computing (Foster, Kesselman, and Tuecke, 2001) has emerged as an important technology for solving large-scale compute-intensive problems where the computational power, storage power, and specialist functionality of arbitrary networked devices can be made available on-demand to any other connected device in a seamless and secure manner. As such, a Grid environment provides an attractive platform for scientific users to solve the most demanding computational and data-intensive problems, because of the great number and variety of resources that a Grid can offer.

In the Grid environment, users and resources are grouped in federations under a common administrative domain, and these federations are commonly referred to as Virtual Organizations (VOs) (Foster, Kesselman, and Tuecke, 2001). A virtual organization is a group of users from multiple institutions who collaborate to achieve a specific goal. A user can take part in different virtual organizations and similarly, a resource can be managed by different virtual organizations. An institution is an administrative domain and has complete control over the resources within its domain. Institutions support a virtual organization and hence allow users, who may belong to different institutions, access to the resources. The Grid middleware is then used to provide a uniform interface at the boundary of the administrative domain to allow interoperation between VO members (Field and Schulz, 2008). Such a concept looks very promising from an overall viewpoint since it provides a platform in which independent institutions and individuals can interact and cooperate in a seamless and secure manner.

However, such approaches in building a large-scale resource-sharing system have several fundamental problems and limitations. First, in the current VO approach, individuals are registered with a universal central authority and their access to a resource is subsequently approved by the VO's administrator (Alfieri et al., 2005). Users need to be members of a VO before they are allowed to run their applications. Furthermore, a VO also requires every resource owner to be registered under its authority. The difficulty of this approach is the management overhead such actions involve. For example, a VO needs a global registration service that validates and approves all users by giving them an access account. Similarly, every resource owner also needs to be registered under a VO before the resource is accessible to all registered users. This imposes the requirement for every user to register with each VO, and every VO in the world also needs to have an account on the resource owner's machine. Considering the millions of users and computers worldwide, this poses a serious management problem because it is impossible for a VO to control this volume of users and resources.

Second, another inherent problem in the current Grid system lies in its centralized approach to scheduling. For example, in the current VO model, there is only one, single, global scheduler controlling all the users and resources (Hauswirth and Schmidt, 2005). The global scheduler is typically controlled by a meta-scheduler (Xhafa and Abraham, 2010). A meta-scheduler is required to maintain and synchronize resource information from each participating local scheduler globally (Choi et al., 2009). Therefore, it constantly needs to monitor the resources state of each participating local scheduler to ensure efficient scheduling. Again, the task of keeping the level of resource state information relevant and correct is enormous.

Faced with a potentially large number of institutions, this poses a number of serious limitations including poor scalability and inadequate support for quality-of-service (QoS), especially from the point of view of urgent job requests (Cencerrado, Senar, and Cortés, 2009). A reservation-based approach can partially address the problem, but such an approach can create severe resource under-utilization due to unused reservations, idle reservation slots, and under-utilized reservations that resource owners are eager to avoid (Park and Humphrey, 2008). For example, an advance resource allocation plan could be ruined or a better choice of resources could be available. Moreover, since advance reservation is statically performed before starting application execution, a job that needs multiple resources would have to wait a long time to have enough resources available to run (Liu, Nazir, and Sorenson, 2007).

As a result, the Grid system suffers from a scheduling bottleneck: scheduling is fairly efficient for best-effort jobs or for small jobs that need only a few processors to execute. However, when the jobs to be scheduled require bounded response times and/or quality of services (QoS), current systems fail to provide acceptable response times as required (Beckman et al., 2006; Cencerrado, Senar, and Cortés, 2009). The result is considerable delay to the jobs starting, which is a clear drawback for urgent distributed applications.

Third, multiple VOs exist throughout different parts of the world (Field and Schulz, 2008). The institutions therefore may participate in different Grid infrastructures. As such, Grid interoperation is needed to bridge these differences and enable virtual organizations to access resources independently of the Grid infrastructure affiliation. Without Grid interoperation, the virtual organization would be limited to only one Grid infrastructure. As different Grids have their own middleware and policies, this introduces another management bottleneck. The challenge of Grid interoperation for multiple virtual organizations and Grid infrastructures further complicates scheduling decisions (Field and Schulz, 2008).

Fourth, resource availability under the VO model is also an issue. Since access to VO resources is regulated by the internal local policies of participating institutions (participants), a VO may have limited capacities due to the limited amount of resources provided by their participants (Vilajosana, Krishnaswamy, and Marquès, 2009). In this sense, during certain periods of time, VOs may become overloaded due to the dynamism of their users or due to the over-consumption of resources by their participants.

Finally, and perhaps most importantly, is the issue of the cost effectiveness and efficiency of global resource-sharing systems. Regardless of the underlying platform (i.e., Cluster or Grid computing), users want to be able to run their applications with reasonable QoS guarantees without making a huge investment in new hardware. The institutions cannot afford to keep multimillion pound infrastructures idle until needed by the applications. Thus, apart from satisfying the application QoS requirements, it is also equally important to justify the investment in building and maintaining the computing infrastructure for running applications. Such a cost justification can be made by maximizing resource productivity (i.e., overall resource utilization) and reducing resource idleness.

Therefore, it is important to leverage the available resources and idle processor cycles to solve a problem more quickly while at the same time maximizing efficiency and reducing the total cost of ownership. However, it is still not perfectly clear whether there is any cost-benefit in building and maintaining a system for the purpose of sharing and running large-scale applications. For instance, cluster and Grid infrastructures have high resource infrastructure costs and they are very expensive to maintain since they incur fixed and operational costs such as those incurred from electricity providers and system administrators (Assuncao and Buyya, 2008). Furthermore, despite the attractiveness of Grid systems which can provide huge and variety computing resources, recent statistical observations (Iosup et al., 2006) have shown that resources at institutions are severely underutilised[1]. One reason for this is that institutions are often forced to 'over provision' their resource infrastructure to ensure that there is sufficient capacity for peak loads and future demand (Duan, Zhang, and Hou, 2003).

To address all of the above mentioned issues, there is a strong need to develop a cost-effective resource management system that can largely self-manage, provide better resource availability, utilization, performance, and scalability at lower cost; and provide the opportunity for incremental investment and immediate return, which is the subject of this thesis.

1.1.1 Cost-Effective Resource Management

This thesis addresses the above mentioned problems by introducing the possibility of managing geographically distributed resources in significantly 'smaller' computing units than are currently managed by a global Grid system. There is no optimal size for such 'smaller' units because that would depend on how the resources are used. Too few units may not be ideal for the execution of massively parallel applications. Such applications would want to use more resources. On the other hand, too many units would simply burden the Virtual Authority (VA) with a large management overhead. The size issue is important, but this is mainly guided by a complex symbiosis between usage pattern and timing. It therefore has no optimal solution.

The smaller units are constructed to form a VA. A Virtual Authority is a collection of resources controlled, but not necessarily owned, by a group of users or an authority representing a group of users. The owner of a resource recognizes only the VA. All permissions, billing and blame for security breaches are associated with the VA. Although individual processes and tasks can always be associated with individual users, this is a matter for the VA and not the resource owner. As far as scheduling is concerned, the VA is independent from competitors and can instead concentrate on managing its own resources.

At first glance, a VA may not appear to be that dissimilar from a VO. However, a VA has fundamental characteristics which differentiate it from a VO. First, a VO consists of a group of users from multiple institutions who collaborate to achieve a specific goal. It integrates services and resources across distributed, heterogeneous, dynamic organizations to allow service and resource sharing when cooperating on the realization of a joint goal. For example, the EUAsiaGrid VO was formed to promote regional collaboration between Asian countries, as well as collaboration with European scientific communities (Codispoti et al., 2010). The EUAsiaGrid VO is made up of 15 institutions from 12 countries.

Unlike a VO, a VA only represents an individual or a group of users and/or applications from a single private institution. A VA is supported by a private system with a pool of resources that it may either purchase or rent. In such systems, termed in this thesis, VA- or rental-based systems, the VA temporarily rents resources from external resource providers worldwide (e.g., personal computers, clusters, Clouds etc.). Once rented, the nodes are moved outside the management of the resource providers (e.g.,

[1]Most Grid production sites, such as DAS-2, Grid500, NorduGrid, AuverGrid, SHarCNET, and LCG, have a system utilisation of less than 60%. In some cases, the system utilisation is well below 40%.

local schedulers) for an agreed period and/or agreement. The size does not matter because the VA has an agreement with resource providers that are willing to rent some of their nodes in times of high demand. The VA has the responsibility to predict its workload with reasonable accuracy within a specific timeframe with the objective to keep its costs (idle resources) down to a reasonable amount.

The creation of a VA maintains the global concept and at the same time introduces a local concept. This concept has many benefits. First, individual users no longer need to be recognized globally. The organization they belong to can hire equipment and can create a local service, to serve its own users and/or applications. This can provide the user with the isolated, customized execution environment needed, and it promotes simplified resource administration. Therefore, scheduling can be carried out with the minimum of difficulty because there are no competing scheduling authorities and the resource pool is limited.

Second, the environment offers an opportunity for users to outsource maintenance to a third party provider. This outsourcing concept has many advantages, for example, users can avoid the difficulty and expense of maintaining their own equipment and can specifically provision resources during peak loads. Moreover, the system can optimize use of the nodes' processing capabilities more efficiently, because resources are managed in significantly smaller units in comparison to global Grids, and at the same time, the system can retain full control. As a consequence, this will result in a much reduced problem and therefore simpler and faster scheduling.

By forming a small VA which can be constructed temporarily from rented resources, the option can be provided of storing resources ready for use under the control of a VA. As such, temporary and unexpected spikes in demand for resources can be accommodated by flexible rental arrangements. This would enable the applications to customize their execution with a set of distinct resource types and could lead to the formation of ideal node configurations based on the applications' workload requirements. The idea is really that a group of applications (or one application) can share resources efficiently without the problems inherent in using a global Grid. It is envisaged that such an approach would offer unprecedented computing power and scalability as well as rapid and significant performance gains over a traditional dedicated clusters and current Grid systems.

A VA also has several distinctive features that differ from the conventional meta-scheduler paradigm; it is built on multi-tier paradigm. The upper tier offers the ability for the applications to interact directly with the VA system using a conventional job submission interface or simple Application Programming Interface (API) calls. Using the API, the calls are handled by the application agent (AA) which resides between the application and the middle tier. The upper tier is built upon our earlier work (Liu, Nazir, and Sørensen, 2009) and it provides support for dynamic resource allocation at an application level to make the application execution benefit from the adaptive, dynamic, and on-demand resource provisioning. It also provides the prospect of removing the application from user control.

The middle tier takes charge of scheduling responsibility whereby the quality of service (QoS) information provided from the AA is used to appropriately schedule applications and jobs to resource nodes based on their requirements, and resource costs. Finally, the lower tier forms a shared pool of ready-to-use compute resources which are rented from resource providers that represent a worldwide pool of computing power. The multi-tier approach essentially differentiates the roles of application management, job/task scheduling, and resource provisioning.

Since a VA is able to control its environment, it is faced with the conflicting goals of renting sufficient computing nodes to provide an adequate level of application satisfaction and of keeping the cost of renting to an acceptable level. For example, on the one hand, renting too many nodes would incur a large management overhead. On the other hand, renting too few nodes would result in long wait times and application quality-of-service (QoS) dissatisfaction. Therefore, there is a need to balance the cost of satisfying user/application demand and the cost of renting computing resources.

The thesis addresses this issue by introducing a costing model that attempts to provide a mechanism for the VA to balance the cost of renting computational resources and the cost of satisfying application QoS requirements. The costing model is based on an economic approach that captures revenues generated from applications and also expenses incurred from renting resources. The rental expenses derive from the need to deploy rented resource nodes from external providers and the operational costs to maintain them. These include the administrative cost (i.e., deployment cost) and the operational cost (i.e., electricity, personnel, floor spaces etc.). Applications express the monetary value of their jobs as the price they will pay to have them run. The gap between the monetary value (revenue), the penalty for not meeting quality of service (i.e., deadline violation), and the resource cost (expenses) to run the job is simply the job's profit. The profit provides a single evaluation metric that captures the trade-off between earning monetary values, the penalty for deadline violation, and paying the rental cost. We demonstrate how such a costing model can be used effectively to provide an adequate level of application satisfaction and keep the cost of renting nodes to an acceptable level. The overall aim of the costing model is to maximize QoS requirements, resource utilization, and scalability.

The provision of cost-aware rental policies is therefore essential for the economic viability of a VA. Resource planning and renting are the responsibility of the VA. A rental policy must provide a set of rules to decide what, when and how many resource nodes to rent in accommodating local demand. The responsibility of a VA is to offer a cost competitive service in order to attract users. It may have an option of choosing the resource providers that best meet users' QoS requirements. It may rent resources based on short- and/or long-term planning and manage these resources according to the needs of the applications. Because it has sole responsibility for the resource in question while is retains management authority, there are no short-term disputes to resolve.

Profit is a standard metric in the commercial world, and it is envisaged that the profit offers a clear, numerical measure for the VA to evaluate its rental decisions. The profit metric addresses the trade-off between the cost of rental and the lost opportunity if customer demand is not met. Effectively, the costing model provides a valuable tool for capacity planning, and it provides the foundation for improving productivity, managing costs, and return on investment for renting resources from resource providers in the presence of 'bursty' and unpredictable application demand.

Outsourcing high-performance computing services often involves service level agreements (SLAs) or contracts that include penalties for poor performance: if the response time is too long, for too many jobs, the VA should earn less, and may even have to pay out more than it takes in. Prior to job execution, the user and the VA may have to agree on a Service Level Agreement (SLA) (Leff, Rayfield, and Dias, 2003) which serves as a contract outlining the expected level of service performance such that the VA is liable to compensate the user for any service under-performance. Therefore, a VA needs to not only balance competing application requirements, but also to enhance the profitability of the provider while delivering the expected level of service performance.

The expected level of service performance typically relates to user experience, such as the completion of a job within its deadline (Yeo and Buyya, 2007). Therefore, the VA must make use of quality of service (QoS) information of a job (i.e., deadline) to determine whether there is an adequate amount of resources available to accommodate the job's deadline, and it should rent additional resources if necessary. However, the problem is that job information does not often reveal sufficient information for the VA to perform long-term planning. This forces the VA to make a rental decision on a reactionary basis because often information regarding a low level of resources or resource unavailability is not known until there is an emergency need. This could result in deadline violations and/or under-utilized resources. Therefore an additional SLA-aware framework is needed that can express the user's desires while not unduly constraining the VA.

This thesis therefore proposes an extension of our initial framework to provide an additional control based on service level agreements (SLAs), or long-term contracts. Such an extension promotes long-term planning, and enables the VA to plan rental decisions in a proactive mode, rather than on a reactionary basis. The extended framework offers the facility for the applications to specify the resource requirements, their expected quality-of-services (QoS), total monetary values, and penalties for the whole period of application execution. This effectively improves rental decisions because long-term capacity planning can be established from the knowledge of the SLA contract. With the incorporation of such a framework, the thesis presents several SLA-aware policies and evaluates them to demonstrate their applicability and effectiveness in improving profits.

1.2 Scope and Assumptions

1.2.1 Quality of Service and Service Level Agreements

As in the human world, users express utility as the budget or amount of real money that they are willing to pay for the service (Buyya, 2002). Real money is a well-defined currency (Lai, 2005) that will promote resource owner and user participation in distributed system environments. A user's budget, monetary or utility value is limited by the amount of currency that he/she has which may be distributed and administered through monetary authorities (Barmouta and Buyya, 2002). In the thesis we focus mainly on the resource allocation techniques to meet the application QoS requirements and their applicability in the context of a VA that rents resources from external providers, rather than owning these resources. On-demand provisioning of computing resources diminishes the need to purchase computing infrastructure since renting arrangements provide the required computing capacity at low cost. Furthermore, since the amount of work the VA receives will vary, the VA can reduce its ‘risk’ by renting the resources it needs from a resource provider, rather than owning them physically.

Resource providers may strategically behave in ways to keep their costs down and to maximize their return on investment. For example, the pricing[2] policies may consider the following question from the provider’s viewpoint: what should the provider charge for the resources in order to benefit from renting out its resources? Since our work focuses on the policy choices made by the VAs to keep their costs down and to maximise their return on investment, other aspects of market dynamics such as pricing policies and incentive mechanism design imposed by resource providers are beyond the scope of this thesis. We will not explore how a market equilibrium may be achieved and how setting certain rental rules and policies would affect the pricing. This thesis does not venture further into other market concepts such as user bidding strategies (Wolski et al., 2001c; Chun et al., 2005) and auction pricing mechanisms (Waldspurger et al., 1992; Lai et al., 2004; Das and Grosu, 2005b).

1.2.2 Costing Model

The costing model makes the assumption of prior knowledge of job deadlines for deadline-driven and/or urgent applications. It is assumed that such information is provided at the application tier by the applications themselves or by agents on their behalf. Furthermore, in some cases, it is further assumed that the knowledge of monetary value is provided. The utility or monetary value is the monetary payment paid by the applications (the price the user is willing to pay) for job execution. In this thesis, we assume that deadline-driven applications or their users are capable of truthfully expressing their monetary values according to their job and task deadlines, and such mechanisms are assumed to be provided by the applications or their users.

1.2.3 Security Issues

Parallel computations that deal with geographically distributed computational resources need to establish security relationships not simply between a client and a server, but among potentially thousands of jobs or tasks that are distributed across different geographical locations or availability zones. The security-related issues have to be addressed before any proposed solutions can be applied in practice. In this thesis, we did not look into security and privacy concerns. We assume the existence of a security infrastructure that authenticates and authorizes users. Such infrastructure should enable an authorized consumer or user to grant access rights to computing resources on a remote site. We also do not concern ourselves here with security policies, confidentiality and data integrity issues. When discussing the prototype implementation of our architecture, we will limit our discussions to the practical strategies that we employ to alleviate the problems of providing secure access to distributed resources behind firewalls.

1.2.4 Adaptive and Reconfigurable Applications

In this thesis, adaptive and reconfigurable applications (Islam et al., 2003; Agrawal et al., 2006; Park and Humphrey, 2008) refer to the distributed or parallel applications that are able to adapt to the various dynamics of the underlying resources. This implies the ability of the application to modify its struc-

[2]We differentiate between pricing nodes and charging for services in the thesis.

ture and/or modify the mapping between computing resources and the application's components while the application continues to operate with minimal disruption to its execution. Such an application has a reasonable knowledge of its inner workings and internal structure to be able to manage its own load balancing techniques to maximize its performance. During runtime, the application itself continuously measures and detects load imbalances and tries to correct them by redistributing the data, or changing the granularity of the problem through load balancing. Effectively, such applications are able to reconfigure themselves, self-optimize, and migrate to adapt to the characteristics of the underlying execution environments. We assume such applications have the following properties, namely, that they: (1) make use of new computing resources during execution; (2) perform internal load balancing; and (3) are resilient to resource failures.

The basic idea behind adaptive and reconfigurable application execution is to make the jobs in the system share the processors equally as much as possible. This is achieved by varying the number of processors allocated to an application during its execution. This means that additional processors may be added to an executing application or job when processors become available. In particular, when resources are added or removed during execution, the application is capable of dynamically performing load balancing on existing and newly configured resources to make the most efficient use of resources. For example, after adding a resource, either process migration or data load balancing may take place to take advantage of the newly added resource.

1.2.5 Cluster, Grid and Cloud Computing

In this thesis, we assume that geographically distributed resources operate within the context of existing dedicated clusters, Grid, and/or Cloud systems. For example, we are able to leverage existing DRM systems such as Condor (Thain, Tannenbaum, and Livny, 2005), Load Sharing Facility (LSF) (Xu, 2001), Portable Batch System (PBS) (Henderson, 1995), and Sun Grid Engine (SGE) (Gentzsch, 2001a) that provide job submission interface with local scheduling and job management functionalities in cluster environments.

For Grid systems, we further assume an architecture comprised of the following components (Xhafa and Abraham, 2010): Grid scheduler, information service, discovery services, security services, and distributed resource managers. In the current Grid set-up, a Grid scheduler (also known as a super-scheduler, meta-scheduler etc.) corresponds to a centralized scheduling approach in which local schedulers or distributed resource managers are used to reserve and allocate resources from multiple administrative domains (i.e., sites, clusters). Most importantly, a Grid scheduler makes job reservations which are necessary for tasks, jobs or applications that have QoS requirements on the completion time or when there are dependencies/precedence constraints which require advance resource reservation to assure the smooth execution of the workflow (Cao et al., 2003). Effectively, the Grid scheduler is in charge of managing the advance reservation, negotiation and service level agreement under a Grid environment. We assume such a Grid architecture when we refer to Grid systems in this thesis. This assumption is realistic and in line with most current Grid systems. For example, the Enabling Grids for E-SciencE (EGEE) Grid, which is currently the world's largest production Grid, operates in this manner. The EGEE Grid employs a broker known as a workload management system (WMS), which acts as a super-scheduler or a meta-scheduler that manages resource information from each participating local scheduler worldwide.

More recently, the term ‘Cloud computing’ (Sullivan, 2009) has been introduced to describe a technology providing elastic and often virtualized distributed resources over the Internet. Cloud computing has evolved from Grid computing, but it provides a feature that allows machines to be customized and dynamically configured (via virtualization (Barham et al., 2003a)) and delivered on demand. Such a feature is important since the trend in distributed computing is to build global resource Clouds composed from many geographically distributed resources.

In this thesis, we are considering a solution where distributed geographical resources (either physical machines and/or virtual machines from Clouds) are created from rented hardware that is under the complete control of the VA. Potentially, the VA will be able to customize the hardware for parallel job execution. For example, the VA should have the option to install its own operating system (via virtualization) or use one of the options offered by the resource owner. The concept of a maintenance-free environment is very attractive to the VA because it may prefer to obtain complete solutions or at least a solution that can be combined with others into a complete system. There may be local facilities involved or all equipment, including long-term resources, may be rented. As such, there is no reason why a general cluster, Grid and Cloud cannot co-exist with such a VA system. The only assumption we make is that the machines are allowed to be taken out of service and given to the VA for customization.

1.2.6 Leasing and Renting Mechanisms

The investigation of the resource rental concept for distributed computing problems is fairly recent (Popovici and Wilkes, 2005; Burge, Ranganathan, and Wiener, 2007), although the underlying technologies to realize such a renting concept have long been proposed via leasing approaches (Waldo, 1999; Gentzsch, 2001b; Chase et al., 2003). In this thesis, we assume the existence of such middleware components and services that can be put to use in implementing a rental-based system. There are not yet any actual implemented rental-based resource management systems that can demonstrate that they work in practice and this is due to the lack of integrated mechanisms for cost-effective rental policies and resource management. Furthermore, none of the resource management systems offer ease of participation, rental decisions assistance with cost efficiency, and on-demand renegotiation for resources at runtime. It is envisaged that effective provision of rental policies can aid actual deployments of a resource management system with the above attractive features. In this thesis, we demonstrate how a rental solution can be applied effectively to promote efficient management of geographically distributed resources.

1.2.7 Computational Resources

In the thesis, we focus our research specifically on the provision and usage of computational resources (i.e., processors and CPUs) for compute-intensive and/or processing-intensive scientific applications. The provision of QoS support for networks, storage systems etc. and other resource types is not explicitly covered and investigated in the thesis. However, the same approach can be adapted to network bandwidth and storage capacity as well. Without loss of generality, for this thesis, we assume one processor (CPU) per node, and an incoming job specifies how many CPUs it needs, and executes one task (process) on each of the allocated CPUs.

1.3 Research Contribution

The goal of this thesis is to propose cost-effective mechanisms that can largely self-manage, provide better resource availability, performance and scalability at lower cost by introducing a system that can be constructed from rented resources, and to propose a set of rental policies and SLA-aware policies that can make such a system a viable alternative to the current distributed computing paradigm. In this model, it is envisaged that users and/or applications can solve distributed computing problems with satisfied execution performance, without having to own or manage the underlying distributed resources.

This thesis makes the following contributions:

1. We disclose a cost-effective framework for supporting high-performance computing applications with minimal infrastructure and resource costs. The approach is based on the multi-tier model that resolves the scheduling and resource management issues by making a distinction between application management, job scheduling, and resource provisioning. The framework has three distinct tiers. The upper tier offers the ability for the end users or the applications to submit jobs using conventional job submission interfaces or to interact directly with the middle tier using simple API calls. The calls are handled by the application agent (AA) which resides between the application and the middle tier to provide a flexible execution environment where the compute capacity can be adapted to fit the needs of applications as they change during execution. The middle tier makes use of the QoS information provided from the AA and appropriately schedules application jobs based on job requirement and resource costs. The lower tier consists of a negotiator that obtains resources from external resource providers and integrates these resources to form a shared pool of ready-to-use compute resources. The multi-tier approach essentially differentiates the roles of application management, job scheduling, and resource renting.

2. A crucial requirement is the ability for the VA to negotiate with and rent additional resources from resource providers dynamically at runtime. To support this requirement, we demonstrate the feasibility of our proposal through the practical implementation of a prototype system. We describe HASEX[3], a proof-of-concept prototype implementation that partially realizes some of the features of a VA system. We define important key features of HASEX and describe how they are implemented in practice. Specifically, we highlight the specific design decisions made that partially address the fundamental requirements of HASEX. The implementation is used to demonstrate, through replicated experiments, that our rental framework supported by the HASEX prototype is comparatively better than the conventional Grid approach.

3. In order to examine the financial impact of building a resource infrastructure for an European research project and an international research project, we perform a cost-benefit analysis of the international EGEE Grid (Berlich et al., 2006) and a dedicated Lawrence Livermore National

[3]HASEX is an acronym for "**H**ao, **A**mril, and **S**øren **EX**ecution system" which was named after its creators.

Lab (LLNL) HPC system. The EGEE Grid is currently a world-leading production Grid across Europe and the rest of the world, whereas the LLNL HPC is a large Linux cluster installed at the Lawrence Livermore National Lab which is being used to run a broad class of applications by high-performance computing users. Through our analysis, we demonstrate that there is a potential cost-benefit in adopting a small private resource system with the ability to rent processing power based on workload demand. This finding has led to the proposal of a VA system that can provide new avenues for agility, service improvement and cost control in comparison to a static Grid resource management system without a rental mechanism in place.

4. With the introduction of a VA approach, the remaining question is then: Is it efficient? The VA needs to quantitatively evaluate its conflicting objectives in order to minimize operating and rental-related costs subject to application satisfaction-level constraints. We present such a costing model, which uses profit as the main evaluation metric. Profit is the trade-off between earning monetary values, penalty for deadline violation, and paying for the rental cost. Based on this costing model, we introduce aggressive and conservative rental policies that operate in a reactionary mode, whereby the system only rents nodes when there is a sudden increase in demand or when the nodes fall to a low level. Taking into account the additional parameters of execution deadlines and virtual currency when performing rental decisions, we further present cost-aware rental policies that incorporate execution deadlines and monetary values when making scheduling and rental decisions. We then explore how these policies can be improved further by taking into account job deadlines, monetary values, system revenue and system profitability, and examine how load, job mix, job values, job deadlines, node heterogeneity, rental duration, node lead time, job sizes, and rental price influence the VA's profit. We also examine the impact of uncertainty in demand, resource availability and resource costs.

   Experimental results show that our VA approach delivers substantially higher profit compared to a static resource system. We show that the proposed rental policies provide significant benefits over a static and a dedicated resource management system, and there is encouraging initial evidence that combining the information on job monetary value, job deadline and system net profit 'on the fly' (i.e., at runtime) when making rental decision leads to higher overall profit increase, and there is good evidence to recommend the use of our rental policies to increase system profitability and maximize application satisfaction for varying workloads and system parameters. Our results provide insight into the benefits of possible optimizations and are a step towards understanding the balance of satisfying customer demand and the cost for renting computing resources. The investigated policies serve as a foundation for improving productivity and return on investment in satisfying demand without a heavy upfront investment and without the cost of maintaining idle resources.

5. We propose an extension of our VA framework to provide an additional control based on service level agreements (SLAs), or long-term contracts. Such an extension promotes long-term planning and enables the VA to plan rental decisions in a proactive mode, rather than on a reactionary basis.

The extended framework offers the facility for the applications to specify their resource requirements, their expected quality-of-services (QoS), total monetary values (TMV), and penalties for the whole period of application execution. This effectively improves the rental decisions because long-term capacity planning can be carried out in advance using the knowledge from the long-term SLA contract. With the incorporation of such a framework, we propose several SLA-aware policies: SLA Rigid, SLA Value-aware, and SLA Profit-aware. We evaluate these policies under varying workload and system conditions to demonstrate their applicability and effectiveness in improving profits.

Through experiments, we show that our SLA-aware policies outperform conventional non-SLA policies across a wide range of conditions. We also make the following observation: the idea of investing and recycling system profit at runtime to accommodate future jobs can actually mitigate the risks of urgent jobs being dissatisfied, and it is a powerful technique for ensuring zero penalty cost in extreme situations of a sudden burst of demand for resources. Furthermore, our proposed SLA Value-aware policy is quite effective as it outperforms all other policies, including the SLA Profit-aware policy across a wide range of conditions. In particular, our evaluation demonstrates that the SLA Value-aware policy can generate significantly higher profit than the SLA Profit-aware policy for workloads with a higher fraction of urgent requests. The observations highlight the need to select an SLA policy according to the ratio of urgent and non-urgent job requests in the workload.

The above contributions are very much complementary in nature. Used in conjunction, the resulting framework presents a unique set of characteristics that distinguish it from existing cluster, Grid, and Cloud systems; however, it still relies on these technologies as its backbone and infrastructure support: the result is an adaptive self-centred approach to collaboration, allowing a VA to construct a dynamic resource management environment and to operate such an environment accordingly in the most cost-effective manner. Unlike centralized approaches or approaches based on fully autonomous behaviour, where independent resource providers operate mostly in isolation, our framework fosters collaboration without compromising site autonomy through its rental-based approach. Furthermore, it is designed to ensure that significant benefit can still be obtained under the current distributed environment without the need for complete cooperation by all resource providers. This thesis will demonstrate how the rental framework can help achieve significant gains in profit, which in turn can provide mutual benefits (i.e., rapid request response and high resource utilization) for both users and resource providers.

The contributions and core content of this thesis have been peer-reviewed (or are currently being reviewed) and have or will be published in the following publications:

- Amril Nazir, Hao Liu, and Søren-Aksel Sørensen. A Cost Efficient Framework for Managing Distributed Resources in a Cluster Environment. In *Proceedings of the 11th IEEE International Conference on High Performance Computing and Communications (HPCC 2009),* Seoul, Korea, 25-27 June, 2009.

- Amril Nazir, Hao Liu, and Søren-Aksel Sørensen. A Rental-Based Approach in a Cluster or a Grid Environment. In *Proceedings of the 10th IEEE International Conference on Scalable Computing and Communications (SCALCOM 2010)*, Bradford, West Yorkshire, UK, June 29-July 1, 2010.
- Amril Nazir, Hao Liu, and Søren-Aksel Sørensen. Service Level Agreements in a Rental-based System. In *Proceedings of the 10th IEEE International Conference on Scalable Computing and Communications (SCALCOM 2010)*, Bradford, West Yorkshire, UK, June 29-July 1, 2010.
- Amril Nazir and Søren-Aksel Sørensen. Cost-Benefit Analysis of High Performance Computing Infrastructures. In *Proceedings of the 10th IEEE International Conference on Service-Oriented Computing and Applications (SOCA 2010)*, Perth, Australia, 13-15 December, 2010.
- Amril Nazir, Hong Ong, S. Thamarai Selvi, Rajendar K, and Mohd Sidek Salleh. IntelligentGrid: Rapid Deployment of Grid Compute Nodes for Immediate Execution of Batch and Parallel Jobs. In *Proceedings of* the IEEE Conference on Open System 2011 *(ICOS 2011)*, Langkawi, Malaysia, 25-28 September, 2011.
- Amril Nazir and Søren-Aksel Sørensen. HASEX: Rental-based Resource Management System for Distributed Computing, *Future Generation Computer Systems the International Journal of Grid Computing and Escience* (submitted).
- Amril Nazir, Bazli Karim, Sidek Salleh, and Ng. Kwang Ming. Method and System for Automatic Deployment of Grid Compute Nodes. Patent PI 2011001637, April 12, 2011.
- Amril Nazir, Sidek Salleh, Thamarai Selvi, Rajendar K, and Fairus Khalid. A Method and System of Extending Computing Grid Resources (Patent Pending).

1.4 Organisation

We end this introductory chapter with an outline of the remainder of this dissertation. The outline of the thesis is as follows:

Chapter 2 introduces the research background and related work from a wide variety of areas related to distributed computing systems.

Chapter 3 sets the stage for the research presented in this thesis by discussing the motivation behind the adoption of the virtual authority (VA) and rental-based mechanisms. We then present fundamental requirements that a virtual authority (VA) system must meet in order to deliver application QoS satisfaction for HPC applications.

Chapter 4 presents HASEX, a proof-of-concept prototype implementation of a virtual authority (VA) system. We describe HASEX's key features and discuss how the implementation is realized.

Chapter 5 evaluates the cost-benefit of a virtual authority (VA) system versus dedicated HPC systems.

Chapter 6 presents resource management strategies that incorporate cost-aware rental policies to make renting decisions.

Chapter 7 introduces the service level agreements management (SLAM) framework that can be used to enhance the overall application satisfactions. We proposed three new SLA-aware policies that make use of SLA information to improve scheduling and rental decisions.

Finally, Chapter 8 concludes the thesis and outlines directions for future work.

Chapter 2

Background

In this chapter, we present a general overview on the topic of distributed computing and resource management including a brief history of distributed computing infrastructures, distributed and parallel application types, job scheduling and resource allocation. We then focus on surveying a wide range of research related to resource management in distributed computing.

2.1 Distributed Computing Infrastructures

Computational science is the field of study concerned with constructing mathematical models and numerical techniques that represent scientific, social scientific or engineering problems and employing these models on computers, or clusters of computers to analyse, explore or solve these models. Numerical simulation enables the study of complex phenomena that would be too expensive or dangerous to study by direct experimentation. The quest for ever-higher levels of detail and realism in such simulations requires enormous computational capacity, and has provided the impetus for breakthroughs in computer algorithms and architectures. Due to these advances, computational scientists and engineers can now solve large-scale problems that were once thought intractable by creating the related models and simulating them via distributed systems.

Distributed computing has long been a focus of both academic research and commercial development, and the field presents a bewildering array of standards, products, tools, and consortia. The basic idea of distributed computing is to group together general-purpose processors to solve advanced and large-scale computations. The practical implementation efforts of distributed systems began in earnest in the early 1980s and have been primarily carried out within academic computer science departments and government research laboratories.

However, any attempt at comparative analysis is complicated by the fact that many of these systems interrelate not as mutually exclusive alternatives, but as complementary components or overlapping standards. In this thesis, distributed computing refers to the use of distributed systems to solve computationally intensive problems. A distributed system consists of multiple autonomous computers that communicate through a computer network. In this section, we provide a brief history of distributed systems and current research developments that are related to distributed computing.

2.1.1 Supercomputer

A parallel computer is "a collection of processing elements that communicate and cooperate to solve large problems fast" (Feitelson and Rudolph, 1995). The main motivation for developing and using such computers is that whatever the performance of a single processor at a given time, higher performance can, in principle, be achieved by utilizing many such processors. Parallel machines are often referred to as 'supercomputers', if the number of processors used is relatively high.

The earliest supercomputers included vector-based array processors, whose defining feature was the capability to perform numerical operations on very large data arrays, and other SIMD (Single-Instruction, Multiple-Data) architectures, which essentially performed an identical sequence of instructions on multiple datasets simultaneously. Multiple-instruction architectures have also been widely adopted, and especially SMPs (Symmetric Multi-Processors), have tended to predominate, although the most powerful supercomputers generally combine features of both.

In the early days, most supercomputers were dedicated to running a vector processor, such as the well-known Cray-1 installed in the 1970s (Smallen et al., 2000). Vector processing, normally with an instruction pipeline, is designed to efficiently handle arithmetic operations on elements of arrays by performing operations on multiple data elements simultaneously. Supercomputers are more commonly designed by massively parallel processing (MPP). A MPP supercomputer is a distributed memory computer system which consists of many individual homogenous nodes, each of which is essentially an independent computer in itself, and in turn consists of at least one processor, its own memory, and a link to the network that connects all the nodes together. Nodes communicate by passing messages, using standards such as Message Passing Interface (MPI) (Gropp, Thakur, and Lusk, 1999).

Installations with tens to hundreds of processors are commonplace nowadays. The "top-500 list" (Feitelson, 2005) (which lists the 500 most powerful supercomputers in the world) is dominated by machines with thousands or more processors and is led by the National Supercomputing Center in Tianjin, China with 186,368 processors. Feitelson (2005) also established that the combined power of the top-500 almost doubles every year and that the combined number of processors is doubled every three years.

2.1.2 Cluster Computing

Years ago, supercomputing (then more commonly referred to as 'High Performance Computing') was dominated by specialized large supercomputer systems found primarily in research centres. As the computing power of small systems increased, the cost/performance ratio changed, and computational workloads moved to workstation-class systems. In particular, with dramatic improvements in the processing power and storage capacity of 'commodity' hardware and growing network bandwidth, much of the focus has shifted toward parallel computing based on loosely-coupled clusters of general-purpose processors, including clusters of networked workstations.

Clusters, built using commodity-off-the-shelf (COTS) hardware components as well as free, or commonly used, software, are playing a major role in redefining the concept of supercomputing and high performance computing. Indeed, such networked workstations consist of resource nodes (e.g., computers, processors, servers, etc.), which are grouped or clustered to perform certain functions. The computers of a cluster are usually connected by a network, such as the Internet or an intranet. Networking is achieved either by using commodity (e.g., Ethernet) or specialist networks (e.g., Myrinet).

The process of allocating applications to individual resource nodes of a cluster is handled by a local scheduler which is also known as local job scheduler and/or Distributed Resource Manager (DRM) (Foster et al., 1999b). The applications or jobs are often categorized either as batch or parallel. Batch jobs require no interaction from a user or an application during execution, and can execute in a serial manner, where each job is fully independent. Parallel jobs, on the other hand, are executed in multiple instances, run on different nodes, and require interaction with each other during runtime. The DRM allocates a job to a node using a resource allocation policy that may take into account node availability, user priority, job waiting time, etc. Examples of popular distributed resource managers (DRMs) include Condor (Thain, Tannenbaum, and Livny, 2005), the Sun Grid Engine (SGE) (Gentzsch, 2001a) and the Portable Batch Queuing System (PBS) (Henderson, 1995). The DRMs typically provide submission and monitor interfaces, enabling users to submit jobs to be executed and to keep track of the progress of execution.

A job will be specified by a job description including a collection of resource requirements, such as processor architecture, memory size and operation system type; the files required for the execution, such as the executables and data files; and the execution parameters such as arguments and environment variables (Foster et al., 1999a). The DRM matches the job requirements with the characteristics of available resources and allocates the most suitable resources to run the job. DRMs provide a resource information interface, providing users with the means to obtain resource information that contains the resource availability, processor architecture, CPU load, operation system, etc.

Existing DRMs such as SGE, Condor, and Sun Grid Engine (SGE) are primarily designed to optimize the overall cluster performance by maximizing the overall job performance and system usage of the cluster. They assume that all job requests are of equal user importance and therefore they do not provide quality-of-service (QoS) support required by different users and applications (Yeo and Buyya, 2007).

2.1.3 Grid Computing

Clusters can mimic parallel supercomputers in many aspects, and are usually geographically compact and under a single administrative domain. However, when dealing with multiple clusters among multiple organizations worldwide, the question of how these resources can be accessed and used as a single integrated service must be answered. Only very recently, advances in wide-area networks have made it possible to aggregate geographically dispersed computing, storage, and network resources across independent sites to execute distributed applications. The execution of such applications in such a wide-area setting has become known as Grid Computing (Foster, Kesselman, and Tuecke, 2001).

The objective of Grid Computing is to pool together resources of all types (e.g., storage, processors, instruments, displays, etc.) anywhere on the network and make them available in a seamless and secure manner to all users. It essentially realizes this objective by connecting multiple resource providers worldwide which are owned by different authorities. Grid computing, as pointed out by Foster, Kesselman, and Tuecke (2001), is a relatively new paradigm but the concepts that underline it are not fundamentally different from Cluster Computing. In the mid-1990s, inspired by the pervasiveness, reliability, and ease of use of electricity, Foster, Kesselman, and Tuecke (2001) began exploring the design and development of an analogous computational power Grid for wide-area parallel and distributed computing. Grids provide a distributed computing paradigm or infrastructure that spans across multiple entities which are known as virtual organizations (VOs) (Foster, Kesselman, and Tuecke, 2001). The Virtual Organization (VO) encompasses users and resources supplied by the different institutions in order to achieve the VO's creation of a common goal. The different organizations in a Grid environment federate into an alliance for a specific goal (e.g., project cooperation, resources rental or application provider). They put in place a Virtual Organization (VO,) which constitutes the shared space between the different institutions. An organization may need to participate in multiple Virtual Organizations according to its needs and thus may have multiple shared spaces with multiple communities (Nasser et al., 2005). Often, a VO will submit all its jobs through the same job manager because, for example, the VO has hired a single technician to handle all its dealing with the Grid or to simplify the accounting of the resource owners. In our model, we assume that each VO can be mapped to a single user (the VO's representative user). Thus, the VOs share with their representative user the limitations imposed by the resource owners.

In Grid Computing, a meta-scheduler (also known as a Grid Broker) (Czajkowski, Foster, and Kesselman, 1999) is the vital part responsible for the distribution of workload and load balancing between multiple DRMs among participating sites (Xhafa and Abraham, 2010). It provides a standardized interface to all DRMs combined with a decision process that enables all Grid users to submit a job with special requirements at a single point and without any global knowledge. For an effective resource allocation decision, the meta-scheduler would need to synchronize resource information (e.g., current resource availability etc.) with all participating local schedulers worldwide on a periodic basis. Based on the workload information collected from the different sites, the meta-scheduler chooses the most suitable site(s) to allocate a job to while the DRM in the chosen site allocates the job to its resource nodes based on its own local scheduling policy. Each participating site via the information service publishes its current load and resource availability information to a central meta-scheduler. The information is used by the meta-scheduler to make well-informed scheduling and load balancing decisions. It coordinates with all participating local schedulers to perform an overall schedule.

The current Grid environment assumes the existence of job schedulers (i.e., DRMs) in different organizations. In fact, a Grid system is typically formed by the 'contribution' of computational resources across institutions, universities, enterprises and individuals which are built up from a number of participating local schedulers at different geographical locations. As such, there are local policies on local schedulers. Due to the different ownership of the resources, there is no full control over individual Grid resources. Therefore, since each local scheduler may employ its own scheduling policy, different local schedulers could, in general, pursue conflicting goals.

In the current Grid environment, a single meta-scheduler is associated with a VO. Faced with a large number of users and resources, the central scheduling point represents a potential bottleneck for jobs needing fast response (Hauswirth and Schmidt, 2005). This is due to the excessive management overhead in synchronizing and maintaining resource states information from fluctuation in load and resource availability. As a result, the meta-scheduler in a Grid environment has issues to scale under a large number of users and resource providers.

The Globus Toolkit (Foster and Kesselman, 1997) is the first standard implementation for Grid systems. Globus enables one to build a uniform computing environment from diverse resources by defining standard network protocols and providing middleware to mediate access to a wide range of heterogeneous resources. Globus provides a number of components to standardize various aspects of remote interaction with DRMs. It also provides a platform-independent job submission interface, the Globus Resource Allocation Manager (GRAM) (Foster et al., 1999a), a security framework, the Grid Security Infrastructure (GSI) (Foster et al., 1998), and information management components, the Monitoring and Discovery Service (MDS) (Zhang and Schopf, 2004). The job submission is described using the Job Description Language (JDL) (*Global Grid Forum*) which is based on Classified Advertisements or ClassAds (Thain, Tannenbaum, and Livny, 2005). The JDL basically consists of a list of key/value pairs that represent the various characteristics of a job (input files, arguments, executable, etc.) as well as its requirements, constraints and preferences (physical and virtual memory, CPU, operating system, etc.). Interactions with the various components of the Globus toolkit are mapped to local management system specific calls and support is provided for a wide range of DRMs, including Condor (Frey et al., 2002), PBS (Henderson, 1995) and Sun Grid Engine (Gentzsch, 2001a). In the toolkit version 4, Globus has been revised to use the web services technologies. Its components have been replaced with newer components and protocols. The new architecture is based on the standard SOA (Service-Oriented Architecture) (Erl, 2005) paradigm in which a web service is used to define methods for describing and accessing component functionalities, and to discover identification of other services.

Furthermore, as part of the standardized effort for job summission, the GridSAM (Lee, Mcgough, and Darlington, 2005) also provides a Web Service for submitting and monitoring jobs managed by a variety of Distributed Resource Managers (DRMs). As a Web Service based submission service, GridSAM implements the Job Submission Description Language (JSDL) Stephen McGough, Lee, and Das (2008), a Global Grid Forum Environments and Group (*Global Grid Forum*) specification aiming to standardize job requirement specification documents. It implements a collection of DRM plug-ins that map JSDL requests and monitoring calls to system-specific calls. These standardised interfaces and tools provide the building blocks for a grid scheduling framework.

The first large-scale production Grid infrastructure was deployed by the Enabling Grids for E-SciencE (EGEE) (Berlich et al., 2006) a European Union (EU)-funded project. The EGEE Grid infrastructure consists of a set of middleware services deployed on a worldwide collection of computational resources, with an extensive programme of re-engineering of the standard Globus middleware that has resulted in a consolidated software stack, gLite (EGEE Project, 2007; Codispoti et al., 2010). The EGEE's gLite services are currently adopted by more than 250 Computing Centres and are used by more than 15,000 researchers in Europe and around the world (Taiwan, Latin America etc.). This makes EGEE the largest Grid in the world. Figure 2.1 shows a simplified system architecture and deployment model for

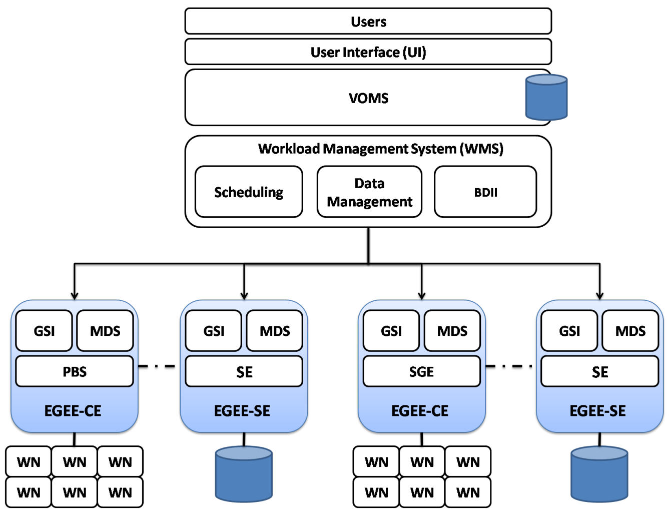


Figure 2.1: Core components of the gLite architecture.

gLite middleware. The core components of the gLite architecture are as follows: a User Interface (UI) which is the access point to the Grid; a VOMS (Virtual Organization Membership Service) which stores a database of users and their roles in the VO. It provides a web-based management interface for updating this information and is also responsible for authenticating users; a Workload Management System (WMS) that chooses a local scheduler (i.e. Compute Element) to submit the job to; a Computer Element (CE) which is a set of computing resources localized at a site (i.e., a cluster, a computing farm); a Berkeley Database Information Index (BDII) which stores information regarding the available resources of a site and the services provided by a site; Worker Node(s) (WN(s)), the resource node(s) where jobs are run; and a Storage Element (SE) which is a separate service dedicated to store data input/output files.

There are also alternative middleware systems to Globus and gLite which include Legion (Grimshaw, Wulf, and Team, 1997), UNICORE (Nicole, 2005), and Nimrod (Buyya, Abramson, and Giddy, 2000). Legion (Grimshaw, Wulf, and Team, 1997) is an object-based, meta-system software project at the University of Virginia. Legion defines a set of core objects that provide basic services, such host objects for job processing and the vault objects for persistent storage. UNICORE is based on multi-tiered architecture that defines different components for each layer. Each layer is represented by the Abstract Job Object (AJO) that contains a description of actions to be performed on different UNICORE sites, regardless the characteristics of the target machines. The AJO also provides the security credentials for accessing distributed resources. Nimrod (Buyya, Abramson, and Giddy, 2000) was originally developed as a modelling simulation tool for steering task farming applications. Nimrod's scheduling is based on the concept of computational economy. The objective is to minimize job completion time within a given deadline budget constraint (time optimization), and to minimize cost within a given deadline (cost optimization). Application developers generate workflows for parametric computing (task farming) and submits their workflows via the Nimrod runtime systems. Legion, UNICORE, and Nimrod middleware systems are deployed in relatively smaller scale in comparison to Globus and gLite middleware.

2.2 Workflow and Advance Reservation

Solving many complex problems in Grids requires the combination and orchestration of several processes (actors, services, etc.). This arises due to the dependencies in the solution flow (determined by control and data dependencies). A workflow (Cao et al., 2003) can take advantage of Grid computing by reserving compute resources from multiple institutions prior to application execution and the job is subsequently executed on the reserved resources.

With the advent of Grid computing, a workflow management engine allows users to specify their requirements, along with the descriptions of tasks[1] and their dependencies using the workflow specification. In the Grid, the QoS is currently supported via advance reservation (AR). AR provides the mechanism that allows distributed and parallel applications to acquire resources from multiple domains from multiple institutions (Islam et al., 2007). It is a process of requesting multiple resources for use at a specific time in the future (Smith, Foster, and Taylor, 2000). The aim of such an AR technique is to reserve resources in advance for an application prior to its execution.

The Highly Available Robust Co-scheduler (HARC) (MacLaren, 2007) offers advanced reservation feature that enables cross-site reservation of time to be made in a consistent manner on multiple compute resources. HARC provides an interface which enables users to interact with the system to create, query, and cancel reservations.

Examples of workflow management engines that support the AR mechanism include the Business Process Execution Language (BPEL) (Andrews et al., 2003), the Vienna Grid Environment (VGE) (Benkner et al., 2004), Pegasus (Deelman et al., 2005), and Nimrod/G (Buyya, Abramson, and Giddy, 2000). The BPEL is currently the standard that supports QoS constraint expression that uses heuristic for scheduling workflow applications. The heuristic approach attempts to assign the task into least expensive computing resources based on assigned time constraint for the local task. The Vienna Grid Environment (VGE) offers a dynamic negotiation mechanism that allows workflow schedulers to negotiate various QoS constraints with multiple resource owners and resource providers. Pegasus uses optimal planning techniques to choose a series of job execution stages that produce the desired output as workflows. It can map and execute individual Directed Acyclic Graphs (DAGs) on a Grid by creating optimal and concrete plans on the fly for the tasks. Nimrod/G offers a cost optimization scheduling policy that aims to minimize cost, while meeting the deadline of the workflow by distributing the deadline for the entire workflow into sub-deadlines for each task. A budget constrained workflow scheduling has also been proposes that uses genetic algorithms to optimize workflow execution time while satisfying the user's budget.

While workflow scheduling is currently a very active area of research, the QoS support mechanism for workflow scheduling is still at a very preliminary stage: the mechanisms for dynamic re-negotiation at application runtime are still not supported to accommodate the processing requirements for adaptive applications. In particular, the characteristics of the Grid environment make the coordination of workflow execution very complex (Cao et al., 2003; Yu and Buyya, 2005). As in other types of scheduling, the

[1]In the context of conventional job submission, the terms "task" and "job" are used interchangeably.

performance and viability is still an issue: it is not very efficient to allocate workflow tasks to distributed resources at multiple institutions because advance reservations have many negative effects on resource sharing and task scheduling in the Grid systems. These negative effects inevitably influence the Grid economy (Buyya, 2002), where resource providers wish to increase the utilization rate of their resources to obtain maximal profits. Therefore, how to mitigate the negative effects brought about by AR becomes an important issue to be addressed.

Smith, Foster, and Taylor (2000) investigate the impacts of advance reservation on performance in terms of mean waiting time and request admission/rejection rate. Their experimental results show that all the three metrics increase when the system is in the presence of a high reservation rate. Their conclusions are repeatedly confirmed by other researchers (Smith, Foster, and Taylor, 2000; F.Heine et al., 2005; Park and Humphrey, 2008). In their studies, they showed that the fixed capability reservation results in a low resource utilization rate, and excessive reservation often leads to a high rejection rate.

Several techniques have been proposed to overcome the limitations of advance reservation. For example, a Globus architecture for reservation and allocation (GARA) (Foster et al., 1999a) architecture was developed, in which the reservation mechanism is enhanced with an intelligent decision model and performance sensors. However, this adaptive reservation architecture is designed for high-end applications with high-bandwidth and dynamic flows, which means that it only adapts to bandwidth reservation for bulk-data transmission.

To increase the resource utilization rate, backfilling (Mu'alem and Feitelson, 2001) is widely used to improve the performance of reservation-based schedulers. Zhang et al. (2001) show that a backfilling mechanism can mitigate the negative effects of advance reservation for those applications with a high ratio of computation to communication. Furthermore, in solving the problem of the high rejection rate incurred when employing advance reservation, Kaushik, Figueira, and Chiappari (2006) propose a flexible reservation window scheme. By conducting extensive simulations, Kaushik, Figueira, and Chiappari (2006) conclude that when the size of the reservation window is equal to the average waiting time in the on-demand queue, the reservation rejection rate can be minimized close to zero.

However, even with many variants of AR techniques, all backfilling variants still rely on a perfect knowledge of job runtime estimates for effective scheduling. The user is still required to provide the runtime estimate information. This poses several issues. Firstly, observations have shown that the estimated-time, which is usually provided by the user, is very inaccurate (Moaddeli, Dastghaibyfard, and Moosavi, 2008). With advance reservation, even a slight amount of inaccuracy in runtime estimation can incur lower resource utilization and higher rejection rates (F.Heine et al., 2005). Secondly, estimating runtime is not always possible because the execution time of jobs can change greatly during application execution. This is especially true for adaptive applications since their computation complexity varies frequently at runtime due to feedback from user actions (Liu, Nazir, and Sørensen, 2009).

2.3 Distributed and Parallel Applications

Historically, supercomputers are used to execute diverse tasks including weather forecasting and climate research (e.g., on global warming), molecular modelling (e.g., of structures and properties of chemical compounds, biological macromolecules, polymers, crystals etc.), various physical simulations (e.g. aeroplanes in wind tunnels, detonation of nuclear weapons, nuclear fusion), cryptanalysis, data mining, and more.

Nowadays, by pooling together independent resource nodes and clusters into Grid systems, scientists have aimed to make use of the increase in the computational power available to tackle larger scientific problems and handle increasingly demanding distributed and parallel applications (Foster, Kesselman, and Tuecke, 2001). Application models are defined in the manner in which the application is composed or distributed for scheduling over Grid resources.

Distributed applications can be broadly categorized as either High Throughput Computing (HTC) or High Performance Computing (HPC) applications (Livny et al., 1997). HTC applications are characterized by a large number of independent small-sized calculations, with moderate resource requirements. Execution of such applications can typically be spread out over long periods of time; days, weeks or even months. On the other hand, HPC applications are compute intensive, and usually require the co-located use of large amounts of resources for a shorter period of time. Jobs will tend to require tighter coupling, and parallel applications fall into this category.

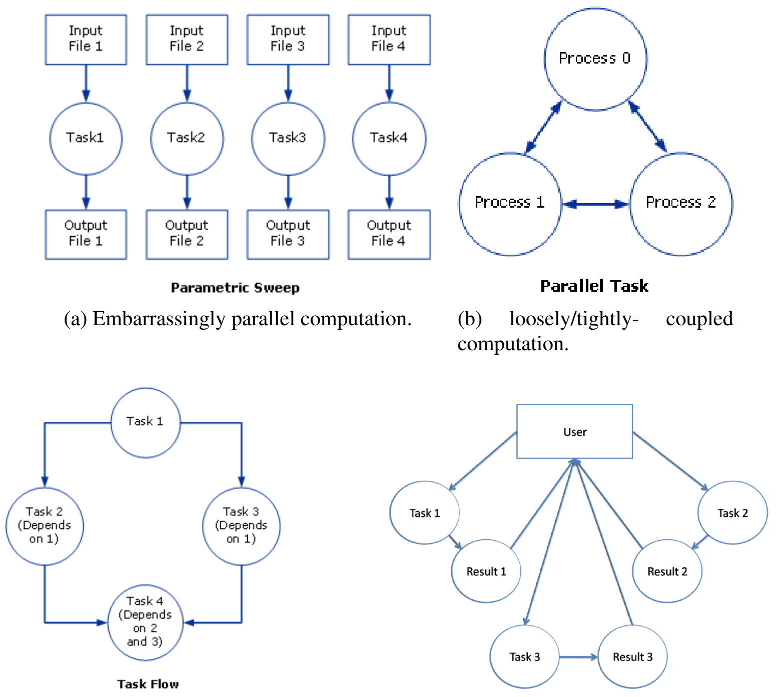
Parallel applications, called 'jobs', are usually composed of a number of independent sequential processes that execute simultaneously, each on a different processor. While they run, the processes communicate and exchange information from time to time, in an effort to complete the task1 as soon as possible. We further differentiate between applications and jobs in this thesis. The concept of 'jobs' in distributed systems is normally used here to refer to distinct workloads submitted to a batch queue.

An application is a higher level concept than a job; although an application could be realized by a single job, it could equally correspond to a coupled simulation, where two jobs belonging to the same application may pass parameters between themselves at runtime.

Parallel applications are typically written using Parallel Virtual Machine (PVM) (Beguelin et al., 1991) or the Message Passing Interface (MPI) library (Gropp, Thakur, and Lusk, 1999). A MPI or PVM library provides interfaces and routines which enable interaction between heterogeneous nodes without being dependent on shared memory in a cluster. Due to recent advances in message passing, the MPI (Message Passing Interface) (Gropp, Thakur, and Lusk, 1999) has become a de-facto standard for communication among processes that model a parallel program running on a distributed memory system. Although PVM and MPI differ in implementations, they are both essentially standards that specify an API for developing parallel applications.

We consider four main categories of distributed and parallel applications and we classify them by their computation and communication structures: (i) embarrassingly parallel computation, (ii) loosely/tightly- coupled computation, (iii) workflow, and (iv) adaptive/interactive applications.

Embarrassingly Parallel Computation. In so-called 'embarrassingly parallel problems', a computation consists of a number of tasks that can execute independently, without communication. Embarrassingly parallel computations can immediately be divided into independent parts which multiple processors can work on simultaneously. Once the work has been allocated, no communication is required between



(a) Embarrassingly parallel computation. (b) loosely/tightly- coupled computation.

(c) Workflow. (d) Adaptive, interactive steering applications.

Figure 2.2: Distributed Application Types.

the processors, so a speedup of n is possible, given n processors. Often the problem can be divided into many more parts than there are processors available to work on the parts, so the maximum speedup possible via parallel implementation is constrained by the number of processors. An example of such an application is the Parameter Sweep Application (PSA) (Buyya et al., 2005), in which the same computation must be performed using a range of different input parameters. The parameter values are read from a set of input files, and the results of the different computations are written to an output file (Figure 2.2a).

Loosely/Tightly-Coupled Communication. Parallel applications have tasks that are often communicated frequently during their execution and these tasks will need to be scheduled and executed in a 'pipeline' fashion (Figure 2.2b). During execution, there will be a set of interdependent tasks that communicate data back and forth. The allocation of these types of applications in a Grid environment involves scheduling multiple individual, potentially architecturally and administratively heterogeneous computing resources so that multiple tasks are guaranteed to execute at the same time in order that they may communicate and coordinate with each other.

Therefore, this resource allocation process is quite different from resource allocation within a 'single' resource, such as a clusters, because in a single-resource environment there is only one local job scheduler or one DRM involved and therefore no resource allocation between multiple administrative domains is required. In a Grid environment where multiple resources are shared between multiple users, coordinated scheduling is accomplished by advance reservation (see Section 2.2).

Workflow. Workflow applications are emerging as one of the most interesting application classes for the Grid. A workflow (Yu and Buyya, 2005) can be described by an acyclic graph, where nodes of the graph represent tasks to be performed and edges represent dependencies between tasks. Workflow Grid applications can also be seen as a collection of activities (mostly computational tasks) that are processed in some order. Usually both control- and data-flow relationships are described within a workflow. For example, the Directed Acyclic Graph (DAG) shown in Figure 2.2c is typically used to represent a workflow application which is composed of a set of tasks where the input, output, or execution of one or more processes is dependent on one or more other processes. The DAG can be read by a Grid workflow engine such as Condor DAGMan scheduler (Deelman et al., 2009). The DAGMan allows users to submit jobs to Condor in an order represented by a DAG and processes the results. The DAGMan scheduler also provides a workflow management engine that negotiates with service providers/resource providers based on the user Quality of Service (QoS) requirement such as task deadlines. Examples of workflow management engines include Business Process Execution Language (BPEL) (Andrews et al., 2003) and Vienna Grid Environment (VGE) (Benkner et al., 2004). The BPEL is currently the standard that supports the QoS constraints expression that uses heuristics for scheduling workflow applications.

Adaptive, Interactive Steering Applications. The resource requirements for most distributed and parallel applications (e.g. embarassingly parallel, loosely/tightly coupled parallel applications) are typically static in nature; their resource requirements do not change during their executions. However, there is an emerging area in the scientific applications field, urgent and on-demand High Performance Computing (urgent HPC), which requires a great capacity of computing resources at a given time (Islam et al., 2003; Park and Humphrey, 2008; Cencerrado, Senar, and Cortés, 2009). Examples of such applications are fluid dynamics (Nivet, 2006), financial analysis (Hu, Zhong, and Shi, 2006), molecular dynamics (Kobler et al., 2007), structural analysis (Alonso et al., 2007) and many others, to name a few.

An adaptive or interactive application may spawn multiple jobs during its execution: it may spawn multiple jobs during execution and the jobs may constantly pass parameters between themselves at runtime. In such a scenario, we consider the application as adaptive because it allows significant changes to its internal dataset structures during execution. Such an application is also interactive because of the need to support human interaction. For example, the application may spawn additional jobs at runtime as a result of computational steering from user actions and/or because of an increase in computational complexity (Figure 2.2d). Moreover, the jobs/tasks may have the requirement that they be completed within specific time deadlines in order to ensure a smooth progression during application execution (Liu, Nazir, and Sorenson, 2007).

For illustration, the Computational Fluid Dynamics (CFD) (Nivet, 2006) simulation often belongs to the adaptive category. CFD is one of the branches of fluid mechanics that uses numerical methods and algorithms to solve and analyse problems that involve fluid flows. One example of a CFD application is the RUNOUT (Sørensen and Bauer, 2003), originally developed to support an EU research project for simulating rock avalanches and sturzstroms. For such an application, individual tasks do not necessarily carry the same workload. Hence, the distribution of tasks among participating processors cannot be determined in advance as batches or workflows. As the model evolves, the load balancing can only be managed by the application itself. During execution, the steering from the user may not only change the numerical values of the datasets, but also spawn additional jobs. This may result in an uneven processing load and interrupt the smooth progression of time. The only option for the application is to request more resources immediately at runtime. A mechanism is therefore needed to handle the urgent demand in maintaining the smooth time progression of application execution. It is envisaged that such a mechanism is needed to provide the capability for applications to request additional nodes and to release their allocated nodes dynamically during execution. However, currently there is a lack of a well-defined interface and the protocol for applications to do so.

More recently, there have been a few research works that have attempted to address this issue. One of them is SPRUCE (Beckman et al., 2006), developed at Argonne National Labs, which is a system to support urgent or event-driven computing on both traditional supercomputers and distributed Grids. Users are provided with transferable tokens with varying urgency levels. During an emergency, a token has to be activated at the SPRUCE portal, and jobs can then request urgent access. The SPRUCE middleware takes care of running the job in a high-priority mode, pre-empting the work that is already running/queued on the machine.

The Application Hosting Environment (AHE) (Zasada and Coveney, 2009) is also a lightweight middleware developed at University College London (UCL) for Grid-based computational science. AHE provides users with the ability to submit applications and run applications in pre-reserved time slots across multiple sites. The reservation of multiple machines at a specified time is achieved using the HARC (Highly Available Robust Co-scheduler) (MacLaren, 2007) system. Used in combination with the HARC, AHE offers the ability to launch web services for computational steering and to launch applications via many different Grid middleware stacks using standards-based interfaces. It also enables access to a common platform of federated computational Grid resources, meaning users can seamlessly interoperate between Grids. AHE has been successfully used to build and integrate campus Grid, science gateway-type portals, and federated computational grid resources on an international scale.

2.4 Job Scheduling and Resource Allocation

Different types of scheduling are found in high performance computing systems. For conventional high performance computing systems such as Cluster computing, the problem of scheduling and resource allocation is not a major issue since the scheduling and resource allocation is restricted in a single administration domain. However, the scheduling and resource allocation in Grid-like systems are more complex as there are multiple administrative domains involved.

The scheduling and the coordination of resources can be performed either in a centralized or decentralized manner. Essentially, both methods differ in the control of the resources and knowledge of the overall resource management system. In the case of centralized scheduling, there is more control over resources: the meta-scheduler has knowledge of the global systems by monitoring the resource state, and therefore it is easier to obtain efficient schedulers. This type of scheduling, however, suffers from limited scalability and has a single point of failure. Therefore, it is not suitable for large-scale resource sharing systems. Another way to organize local job schedulers is hierarchically, which allows one to coordinate different schedulers at a certain level. In this case, schedulers at the lowest level in the hierarchy have knowledge of the resources. Although this scheduler types scales better and is more fault tolerant than centralized schedulers, it still suffers from lack of scalability and fault tolerance.

In a decentralized approach, there is no central entity controlling the resources. The autonomous Grid sites make it more challenging to obtain efficient schedulers. In decentralized schedulers, the local schedulers (i.e., DRMs) play an important role. The scheduling requests that are generated by local users or other Grid schedulers, are sent to local job schedulers that manage and maintain the state of the job queue. This type of scheduling is more scalable for real Grid systems of large scale, but applications may suffer performance problems (e.g., QoS violations) because the resource provisioning system cannot guarantee that the right type and the right amount of resource nodes can be made available immediately upon request.

Due to these issues, today's Grids basically provide merely a 'best-effort' service to their applications. Best-effort systems are characterized by a lack of service level guarantees, and the quality of service they provide may vary over time due to workloads, resource availability, etc. However, this is unacceptable if Grids are to be used for HPC applications with strict QoS requirements.

Therefore, the system must satisfy urgent jobs (i.e., interactive jobs) that need to complete within a given period of time (Zhang et al., 2000) as well satisfy best-effort jobs, whose requirement is to complete the execution as early as possible (i.e., to minimize response time). Best-effort jobs are relatively insensitive to the performance they receive from the Grids and they do not require performance bounds. On the other hand, urgent jobs have stricter quality-of-service (QoS) requirements such as bounded response times or guaranteed throughput (Zhang et al., 2001).

Advance reservation is currently the standard approach to reserve resource nodes from multiple domains in advance to ensure that the QoS requirements can be met. This reservation-based approach distinguishes significantly from other best-effort- or priority-based approaches, because it guarantees the exclusive availability of the required resources in advance (Engelbrecht and Benkner, 2009). However, as we have discussed previously (see Section 2.2), advance reservation is not a viable solution because it results in a low resource utilization rate, and excessive reservation often leads to a high rejection rate. These negative effects inevitably influence the resource cost, where resource providers wish to increase the utilization rate of their resources to obtain maximal profits (Xiao and Hu, 2009). Hence, mitigating the negative effects brought about by advance reservation becomes an important issue to be addressed.

2.4.1 Scheduling Distributed Data-Intensive Applications

Distributed applications are typically data-intensive applications since they typically consist of multiple stages of processing performed in a pipelined manner (Kola et al., 2004). The data transfer between stages is mostly done via pipes and files. They often require a large amount of input data and they also generate a large volume of output data which further require a large storage capacity. Thus, the large amount of input needed and output data generated for each job would suggest that job scheduling policies should take data locality into account. Furthermore, intelligent communication and data management strategies like replication and caching are useful to reduce the latency involved in remote data access.

The thesis considers data locality/storage issues as a secondary issue although there is significant work that recognizes the importance of data location in Grid scheduling. Thain et al. (2001) describe a system that links jobs and data by binding execution and storage sites into I/O communities that reflect physical reality. Jain et al. (1997) proposed a set of heuristics for scheduling I/O operations so as to avoid transfer bottlenecks in parallel systems. The Execution Domains (Basney, Livny, and Mazzanti, 2000) also provide a framework which defines an affinity between CPU and data resources in the Grid, so applications are scheduled to run on CPUs which have the needed access to datasets and storage devices. The framework also includes domain managers, agents which dynamically adjust the execution domain configuration to support the efficient execution of Grid applications.

Building on these research works, Ranganathan and Foster (2003) proposed dynamic data replication and scheduling policies that take into account data locality, storage, current loads, network conditions and network topology. Venugopal, Buyya, and Winton (2006) further proposed a deadline-driven scheduling algorithm for data-intensive scientific applications. Casanova and Dongarra (1996) also described an adaptive scheduling algorithm for parameter sweep applications. Buyya et al. (2005) proposed algorithms in Grid environments that takes locality of data and data storage issues into account. Data Grid scheduling algorithms can be categorized according to whether they exploit the spatial or temporal locality of the data requests (Venugopal, Buyya, and Ramamohanarao, 2006). Spatial locality is the locating of a job in such a way that all the data required for the job is available on data hosts that are located close to the point of computation. In contrast, temporal locality exploits the fact that if data required for a job are close to a compute node, subsequent jobs which require the same data should also schedule to the same node. Spatial locality can be described as ‘moving computation to data’, while the temporal locality can be described as ‘moving data to computation’. The standard approach which couples data replication to job scheduling is based on the temporal locality of data requests.

To resolve the spatial and temporal locality issues, Palmieri (2009) and Coveney et al. (2010) explored how dedicated network links can be used to provide high quality of service, high bandwidth, and low-latency network interconnects between resource nodes to enable the execution of large-scale distributed data-intensive applications across geographically dispersed administrative domains. Their performance tests have shown that dedicated network links can provide high performance for running large-scale distributed scientific applications that require simultaneous, interactive access to compute- and/or visualization resources in supporting interactive visualization and steering capabilities.

2.5 Economic-based Distributed Resource Management

Current Grid computing systems (e.g., Globus, gLite) and conventional distributed resource managers (e.g., LSF, PBS, SGE, Condor) operate on a zero settlement basis. Users give their resources for free and researchers connect across their networks to utilize them. As such, availability of computational resources is unpredictable and jobs are allocated on a first come first served (FCFS) basis. Recently, there have been significant research works that have attempted to address the resource allocation problems using economic-based computing and/or market-oriented approaches (Wolski et al., 2001c; Chun and

Culler, 2002b; Andrzejak et al., 2002; Sherwani et al., 2002; Kleban and Clearwater, 2004; Dube and Parizeau, 2008). These approaches have led to the development of different negotiation and agreement strategies which can be implemented to support the QoS requirements for a broad range of parallel applications.

The basic idea of economic-based or utility-based scheduling schemes is to make use of monetary values and cost-driven information to resolve the problems inherent in conventional advance reservation, allowing cost-effective scheduling based on market economics where the user specifies how important the job is (Dube and Parizeau, 2008). Since users are forced to assess their job, the scheduling algorithm gets more insight into which jobs to prioritize.

Incorporating utility computing into resource management systems is a subject of active research (Rappa, 2004; Yeo and Buyya, 2005; Yeo and Buyya, 2006; Dube and Parizeau, 2008; Costa et al., 2009). As a result, several economic-based resource management systems have been proposed by researchers. Widely known examples of economic-based resource management systems include Spawn (Waldspurger et al., 1992), REXEC (Chun and Culler, 2000), Tycoon (Lai et al., 2004), and Shirako (Irwin et al., 2006), and SHARP (Fu et al., 2003).

Spawn (Waldspurger et al., 1992) is a market-based computational system that runs on a network (mostly LAN) of idle heterogeneous high-performance workstations running Unix. Spawn employs a Vickrey, sealed-bid, second price, auction mechanism. The term 'sealed' refers to the notion that bidding agents have no access to other agents bid information and 'second-price' means that the highest bidder will pay the price offered by the second-highest bidder. In such a system, if there is a unique bid for a resource, the bidder gets it for free, in the absence of competition.

REXEC (Chun and Culler, 2000) implements bid-based proportional resource sharing where users compete for shared resources in a cluster. Resources in REXEC are allocated proportionally based on costs that competing users are willing to pay for a resource. Costs are defined as rates, such as credits per minute, to reflect the maximum amount that a user wants to pay for using the resource. REXEC allows an application to directly manage the execution of its jobs on the selected cluster nodes. REXEC also supports the execution of both sequential and parallel programs. However, REXEC does not allow applications to modify their initial QoS specifications after job submission, which can be a significant disadvantage for adaptive, interactive HPC applications.

Developed at HP Labs, Tycoon (Lai et al., 2004) is a market-based distributed resource allocation system based on proportional share allocation. Along with other market-driven schedulers, it forces users to differentiate the value of their jobs in order to increase overall utility values. Tycoon implements continuous bids where users express a bid for a resource on a given processor at a given price point for a duration. On each processor, the auctioneer calculates each bid and allocates resources in proportion to bids. While this continuous bidding scheme alleviates the user's burden of posting bids on the market for each job, it does not guarantee resource allocation prior to execution, a major impediment to most users with computational deadlines.

SHARP (Fu et al., 2003) is architecture for secure resource peering and sharing across participants and trust domains. Its stands for Secure Highly Available Resource Peering and is based around timed claims that expire after a specified period. SHARP implements tickets that are a form of soft claim that does not guarantee resource availability for the specified timeframe. Tickets enforce probabilistic resource allocation in a manner similar to an aeroplane ticket reservation that only becomes ‘real’ once the boarding pass is printed. This allows SHARP system managers to overbook resources and to defer hard claims or lease allocation until execution. SHARP also presents a very strong security model to exchange claims between agents, either site agents, user agents or third party brokers, that achieves identification, non-repudiation, and encryption.

One of the most interesting works investigating the realms of a true Grid Economy is the G- commerce project (Wolski et al., 2001b). The project considers an unregulated market system where consumers and providers act in their own interest for the system to eventually reach an equilibrium. Mirage (Chun et al., 2005) is also another economic-based resource allocation system which is based on repeated combinatorial auctions (a form of double auction expressing tradeoffs between various options) for a testbed of 148 wireless sensor nodes.

Furthermore, Libra (Sherwani et al., 2002) is a recent well-known market-based cluster DRM that provides hard QoS support for sequential and embarrassingly parallel batch jobs. The Libra system extends the Sun Grid Engine cluster management system to offer a market-based economy driven service for managing batch jobs on clusters by scheduling CPU time according to user utility rather than system performance considerations. It adopts the commodity market economic model that charges users using a pricing function. The main objective of Libra is to maximize the number of jobs whose QoS requirements can be met. Libra considers both time and cost QoS attributes by allocating resources based on the deadline and budget QoS parameters for each job. Libra currently assumes that submitted jobs are sequential and single-task. Users can express two QoS constraints: deadline by which the job needs to be completed and budget which the user is willing to pay. The QoS constraints cannot be updated after the job has been accepted for execution. Hence, only sequential and embarrassingly parallel batch type jobs are supported.

Economic-driven scheduling is based on the economy concept that provides a mechanism for regulating demand and supply of resources, and calculates pricing policies based on these criteria. Recent work on economic-based scheduling policies includes that of (Yeo and Buyya, 2007), which proposes a pricing model for economic-driven management and allocation of resources on a cluster. In such a model, users are assigned an initial allocation of money, a currency they then use to bid for CPU time. Yeo and Buyya (2007) demonstrated that such economic-centric scheduling techniques produce an enhanced resource allocation compared to traditional system-centric schedulers targeting global optimization. This improvement is mostly due to the competitive nature of the system, where the law of supply and demand regulates selfish behaviour of agents, whether they be consumers or providers. Chun and Culler (2002b) further proposed the FirstPrice scheduling policy which has been shown to improve the standard Shortest Job First (SJF) strategy by as much as 14 times for highly parallel workloads. Irwin, Grit, and Chase (2004) further present the FirstReward economic-based scheduling approach by incorporating an urgency component related to the rate at which the value of a job decreases as it waits in the queue. The FirstReward scheduling algorithm aims to balance the risk of keeping a job waiting in the queue with the reward of serving it immediately.

Finally, Wu, Zhang, and Huberman (2005) presented a solution to the truth revelation problem in reservations by designing option contracts with a pricing structure that induces users to reveal their true likelihoods that they will purchase a given resource. Truthful revelation helps the provider to plan ahead to reduce the risk of under-utilization and overbooking. In addition to its truthfulness and robustness, the scheme can extract similar revenue to that of a monopoly provider who has accurate information about the population's probability distribution and uses temporal discrimination pricing. This mechanism can be applied to any resource that exhibits bursty usage, from IT provisioning and network bandwidth, to conference rooms and airline and hotel reservations, and solves an information asymmetry problem for the provider that has traditionally led to inefficient over or under provision.

2.6 SLA Based Resource Management

An SLA (Leff, Rayfield, and Dias, 2003) can be described as a legally binding contract between the parties involved. The agreement relates to a transaction for the provision of a service; as a result, the parties of an SLA can be distinguished as either the providers or the consumers of a service. The terms of the SLA describe the expected level of service within which the service will be provided; these have been agreed between service providers and service consumers. The agreements are typically embedded in the form of Quality of Service (QoS) constraints, such as time deadlines, price etc. The use of time-related constraints suggests some similarities with real-time scheduling.

There has been relatively little research on the use of SLAs for the job scheduling of parallel applications. In fact, the majority of work on SLAs relates to Web Services and networking (Duan, Zhang, and Hou, 2003). Initial work on SLA is presented by Czajkowski et al. (2002) which proposes a generalized resource management model where resource interactions are mapped onto a well-defined set of platform-independent SLAs. The model is based on the Service Negotiation and Acquisition Protocol (SNAP), which is embedded into the Globus Toolkit. However, the most relevant field to the problem discussed in this thesis is real-time scheduling where scheduling decisions, as with SLAs, are time critical. As such, several techniques have been proposed to model negotiation, ranging from heuristics to game theoretic and argumentation-based approaches (Jennings et al., 2001; Chhetri et al., 2006; Qiang He, 2006; He et al., 2009). Recent work on SLAs also investigates the economic aspects associated with the usage of SLAs for service provision such as enforcing charges for successful service provision and penalties for the failure to meet SLAs (Rappa, 2004; Buyya et al., 2005).

In the context of SLA-based job scheduling, the idea of using SLAs is to alter the approach used to perform job scheduling. More recent works have extended the SLAs to make use of economic-based approaches which are highly relevant to the scope of this thesis. Some of the few resource management systems that provide economic-based SLA support are Cluster-On-Demand (Chase et al., 2003), CSF (*Community Scheduler Framework (CSF)* 2010), VIOLA MetaScheduling Service (MSS) (Eickermann et al., 2007), and ASKALON (Fahringer et al., 2005).

Cluster-On-Demand (COD) (Chase et al., 2003) allows the cluster manager to dynamically create independent partitions called virtual clusters (vclusters) with specific software environments for each different user groups within a cluster. This in turn facilitates external policy managers and resource brokers in the Grid to control their assigned vcluster of resources. In COD, the task assignment among various cluster managers adopts the tendering/contract-net economic model (Yeo and Buyya, 2007). A user initiates an announcement bid that reflects the user's valuation of the task and this is transmitted to all the cluster managers. Each cluster manager then considers the opportunity cost (gain or loss) of accepting the task and proposes a contract with an expected completion time and price. The user then selects and accepts a contract from the cluster manager which best fits the user's requirements. Tasks to be executed are assumed to be single and sequential. For each task, the user provides a value function containing a constant depreciation rate to signify the importance of the task and thus the required level of service. The value function remains static after the contract has been accepted by the user. Tasks are scheduled externally to cluster managers in different administrative domains.

CSF (*Community Scheduler Framework (CSF)* 2010) is an open source implementation of an OGSA-based meta-scheduler developed by Platform Computing Inc. It provides a set of modules that can be assembled to create a meta-scheduling system that accepts job requests and executes them using available Grid compute services (Foster and Kesselman, 1997). Grid-level scheduling algorithms can, for example, include scheduling across multiple clusters within a VO, co-scheduling across multiple resource managers, scheduling based on SLAs and economic scheduling models.

The VIOLA MetaScheduling Service MSS (Eickermann et al., 2007) is also a plug-in system that allows a meta-scheduler to negotiate SLAs with the DRMs systems. It is implemented as a Web Service receiving a list of resources pre-selected by a resource selection service. The MetaScheduling Service negotiates a potential start time for the job and subsequently requests reservation of the computational resources.

ASKALON (Fahringer et al., 2005) is a Grid project of the Distributed and Parallel Systems Group at the University of Innsbruck. The objective of ASKALON is to simplify the development and optimization of applications that can utilize a Grid for computation. ASKALON is used to develop and port scientific applications as workflows in the Austrian Gridsation project. The developers designed an XML-based Abstract Grid Workflow Language (AGWL) (Fahringer, Qin, and Hainzer, 2005) to compose job workflows. In ASKALON, SLAs can be made with the resource for a specified timeframe by using the GridARM Agreement package (Siddiqui, Villazón, and Fahringer, 2006). The GridARM package provides a capability to reserve a defined resource capacity such as the number of requested CPUs within the agreed timeframe.

There has also been a significant amount of research to standardize the efforts of SLAs. In particular, work within the Grid Resource Allocation Agreement Protocol (GRAAP) Working Group of the Open Grid Forum (*Global Grid Forum*) (and earlier within its predecessor, the Global Grid Forum) has led to the development of the WS-Agreement (WS-A) (Andrieux et al., 2006), a specification for a simple generic language and protocol to establish agreements between two parties. The goal of the GRAAP

Working Group is to produce a set of specifications and supporting documents which describe methods and means to establish Service Level Agreements between different entities in a distributed environment. The group especially focuses on the usage of SLAs and the WS-Agreement, an SLA format and Web Services protocol specified by the GRAAP, in distributed systems, Grids and Clouds.

Apart from WS-A, WSLA (Keller and Ludwig, 2003) is also another similar framework developed by IBM for specifying and monitoring Service Level Agreements (SLA) using Web Services. The framework aims to measure and monitor the QoS parameters of a Web Service and reports violations to the parties specified in the SLA. In a Web Service environment, services are usually subscribed to dynamically and on demand. In this environment, automatic SLA monitoring and enforcement helps to fulfil the requirements of both service providers and consumers. The WSLA provides a formal language based on XML Schema to express SLAs and a runtime architecture which is able to interpret this language. The runtime architecture comprises several SLA monitoring services, which may be outsourced to third parties (supporting parties) to ensure a maximum of objectivity. The WSLA language allows service customers and providers to define SLAs and their parameters and specify how they are measured.

2.7 Virtualisation

Virtualization has re-emerged in recent years as a compelling approach to increasing resource utilization and reducing IT services costs. The common theme of all virtualization technologies is the hiding of the underlying infrastructure through the introduction of a logical layer between the physical infrastructure and the computational processes. Virtualization takes many forms: 'system virtualization', also commonly referred to as 'server virtualization', is the ability to run multiple heterogeneous operating systems on the same physical server. With server virtualization a control program (commonly known as 'hypervisor' or 'virtual machine monitor') is run on a given hardware platform and simulates one or more other computer environments (virtual machines). Each of these virtual machines, in turn, runs its respective 'guest' software, typically an operating system, which runs just as if it were installed on the stand-alone hardware platform. Additional forms of virtualization include 'storage virtualization' and 'network virtualization', which provide the ability to present a logical view of the storage and network resources, respectively, which is different than the underlying physical resources.

Xen (Barham et al., 2003b) and KVM (Kivity, 2007) are prime examples of hypervisors or virtual machine monitors. Both Xen and KVM are used by the Berkeley NOW (Culler, 1997) and High Performance Virtual Machines (HPVM) (Chien et al., 1997) to aggregate the computing power of commodity machines (interconnected using high-performance communication techniques) and to create a virtual high- performance supercomputer. This involves the partition of physical servers into multiple virtual service nodes to host different services, each running on a guest OS. Another interesting use of virtual machines (VMs) is In-VIGO (S. Adabala and Zhu. 2005). In-VIGO is a distributed environment supporting multiple applications which share resource pools. When a job is submitted, a virtual workspace is created for the job by assigning existing VMs to process it. VioCluster (Ruth, McGachey, and Xu, 2005) is also a similar computational resource-sharing platform that enables virtual domains to safely exchange machines between them without infringing on the autonomy of either domain. It also allows a cluster to dynamically grow and shrink based on local resource demand. Under this system, the administrative privileges of both the renting and lending clusters are maintained: cluster administrators are able to configure rented machines as required, while not granting root privileges to others making use of their nodes.

Advanced virtual networking facilities are also supported by PlanetLab (Peterson et al., 2006), VIOLIN (Jiang and Xu, 2004), and VNET (Sundararaj and Dinda, 2004). With such facilities, virtual machines can be allocated with real IP addresses from the host network. PlanetLab (Peterson et al., 2006) uses a technique to share a single IP address among all virtual machines on a host by controlling access to the ports. These techniques allow virtual machines to connect to a network to create a virtual network. On the other hand, VIOLIN (Jiang and Xu, 2004) is capable of creating isolated virtual environments including virtual switches, routers for the virtual machines (VMs). VNET (Sundararaj and Dinda, 2004) also provides a mechanism to connect remote virtual machines from a local physical LAN with the same network configuration, for example, the same IP address, routes, etc. It enables virtual network cards to map onto another network, which also moves the network management problem from one network to another. This allows a virtual machine to be migrated from site to site without manual network reconfiguration.

2.8 Cloud Computing

More recently, Cloud Computing has become the latest buzzword for the new form of distributed computing. Similar to Grid computing, Cloud computing provides computation, software, data access, and storage services that do not require end-user knowledge of the physical location and configuration of the system that delivers the services. Cloud computing is motivated by the advancement of virtualisation and virtual networking technologies. Providers of compute cycles in the cloud, such as Amazon EC2 (Amazon, 2009) or the Science Clouds (Sullivan, 2009), enable users to acquire on-demand compute resources by renting them.

Based on virtualization technologies such as Xen and KVM, several cloud management resource management systems have been developed, presenting the user with the ability to deploy several VMs over a cluster of physical machines, from a centralized management node, thus building what is commonly referred to as a 'VM-based' cloud system. Examples of such cloud resource management systems include Eucalyptus (Nurmi et al., 2008), OpenNebula (Sotomayor et al., 2009) and Nimbus (Martinaitis, Patten, and Wendelborn, 2009).

The self-service nature of cloud computing allows end users to create elastic environments that expand and contract based on the workload and target performance parameters. Resources can be rented dynamically and renting provides support for on-demand allocation and de-allocation of computing resources with minimal customer capital expenditure since the infrastructure is owned by the resource owners and/or providers. Users can merely access or rent computing resources and deploy services on them. Users need not have knowledge of, expertise in, or control over the technology infrastructure 'in the cloud' that supports them. This is particularly suitable for customers who have workloads that have tremendous spikes and valleys. For example, the workloads for a webserver, or mailserver can be highly demanding at times but often they do not need to run all the time. Cloud computing therefore can be a feasible solution.

Cloud computing is specifically designed to cater for commercial applications. Recently, Amazon EC2 (Amazon, 2009) provides resizable compute capacity in the cloud with the flexibility to choose from a number of different virtual machine types to meet HPC user needs. However, among the most challenging problems posed by the system are resource coordination and scheduling, that is, how to manage cloud resources and schedule them for running distributed and parallel applications.

2.9 Chapter Summary

In this chapter, we have provided an overview of distributed computing, including supercomputing, Cluster computing, Grid computing, distributed/parallel applications and job scheduling, and related work on distributed systems which includes economic-based computing and service level agreements. The background knowledge described in this chapter sets the stage for our work.

Current distributed computing was developed from the traditional supercomputing community. Nowadays, it is still rooted firmly in 1970s supercomputer culture with its batch management. This is reflected by the domination of queuing system-based DRMs, batch-type applications, and the static non-adaptive job submission approach. Many limitations have been inherited from the batch computing culture. The limitations make the computing model more suited to support traditional 'batch computing' or 'high throughput computing', where users are not sensitive to their application performance and the response time sought. This model lacks the mechanism needed to support QoS requirements for distributed large-scale adaptive and interactive applications.

Most importantly, it is observed that current distributed systems suffer heavily from low resource utilization and scalability issues due to the high overhead of maintaining a global system under a large number of users, applications and large pool of resources. Although recent work in economic-based resource management and SLA-based resource management can offer partial solutions to these issues, there is still lack of support for basic requirements such as autonomous decentralized control, rapid provisioning and fast access to resources, and the scalability required without driving up the cost and complexity. Therefore, we conclude that there is a strong motivation to resolve inadequate QoS support, low resource utilization, and scalability issues in large-scale resource sharing environments.

In the next chapter, we will clarify the motivation and requirements for cost-effective resource management and further develop the motivation for the adoption of the approaches presented in this thesis.

Chapter 3

Motivation and Requirements

Distributed computing infrastructures are increasingly central to information infrastructure worldwide. The underlying technologies and services that connect multiple resource providers (e.g., resources, clusters, sites, organizations, etc.) worldwide have become fundamental tools for scientific research, industrial production and both government and industry decision making. A fundamental characteristic of such systems is that it is controlled by a single global scheduler (i.e., a meta-scheduler) or several layers of global schedulers in which scheduling decisions are integrated from many independent resource providers, each of which desires to retain control over its own resource.

Considering a population on the scale of millions of end users and numerous resource providers, it becomes a challenge in managing the complexity that derives from the quantity of end users, resources, and numbers of concurrent and possibly conflicting actions. The result is that the scheduling of high performance scientific applications across multiple providers can be poor from both user and resource management perspectives. This chapter explores the issues and potential limitations associated with current distributed computing infrastructures. Then, having explored these issues and limitations, we present the fundamental requirements that the resource management system must meet in order to address and resolve these issues and limitations.

3.1 Introduction

From the middle of the 20th century, unprecedented complex scientific research such as High Energy Physics (HEP) has been looking for an extremely large amount of computing resources. Supercomputers have traditionally been used to provide such immense processing capability, but in recent years, it has been more feasible to build ‘supercomputers’ by connecting thousands of cheap workstations to form a large, dedicated, high performance computing (HPC) system. The most well-known example is Beowulf (Ridge et al., 1997) which was a supercomputerlike system created from a collection of a number of desktop PCs connected by a high-speed network. The advent of high-speed networks has also enabled the integration of resources which are geographically distributed and administered at different domains. In the late 90’s, Foster and Kesselman (1997) proposed the plug in the wall approach known today as Grid computing, which aimed to make global geographically dispersed computer power as easy to access as an electric power grid.

From this earlier effort, The Enabling Grids for E-science (EGEE) Grid, managed through the European Grid Initiative, has built an e-science resource infrastructure accessible to scientists and engineers from more than 260 institutions in 55 countries. The infrastructure executes an average 330,000 jobs per day, consists of 150,000 processing cores, and offers close to 28 Pbytes of storage (Lingrand, Glatard, and Montagnat, 2009). The system began with applications from only the high-energy physics and life sciences, but it now integrates applications from nearly every scientific field.

In the last few years, cloud computing (Sullivan, 2009) has been emerging as a technology that provides elastic and often virtualized distributed resources over the Internet. Cloud computing provides a cost-effective and rapid way to increase capacity or add processing capabilities on demand without investing in new infrastructure. While grid computing also offers a similar facility for computing power, cloud computing is not restricted to HPC. A cloud can offer many different services, e.g., web hosting, database transactions, etc.

3.2 Unresolved Issues with Current Resource Management Infrastructures

Cluster and Grid computing have been predominantly focused on the resource sharing infrastructure for the provisioning of compute and data resources. On the other hand, the focus of Cloud computing is to offer a cost-effective and rapid way to increase capacity or add processing capabilities on demand without investing in new infrastructure. It is envisaged that these technologies can be integrated together to meet the increasing requirements for solving complex HPC. However, although the technologies can complement one another, there is still a lack of the necessary mechanisms to harness the power of geographically distributed resources for a wide range of distributed applications with quality-of-service (QoS) requirements. In this section, we discuss the shortcomings of current resource management infrastructures in more detail.

3.2.1 Scalability Bottleneck

The scale and complexity of grid systems is constantly increasing. The largest current grid infrastructures, such as the EGEE Grid (Berlich et al., 2006), consist of hundreds of resource centres located worldwide providing tens of thousands of CPUs. It is expected that future Grids will be on an even greater scale. As the number of sites, nodes, and users in the Grid grows, efficient job scheduling becomes increasingly difficult and traditional approaches to job scheduling gradually become inadequate.

Centralized schedulers, often used in single clusters, do not scale and introduce policy problems since they require a single authority to coordinate the resources of the entire Grid. Hierarchical schedulers (Xavier, Lee, and Cai, 2003), more commonly used in larger Grids, only partially address the challenge since they rely on meta-schedulers which also have scalability and reliability limits. In such a centralized scheme, a central meta-scheduler collects all job requests and uses resource utilization information to schedule computation and communication tasks efficiently (Foster and Kesselman, 1997; Field and Schulz, 2008). In order to schedule efficiently, the meta-scheduler needs to keep track of information on the resource availability from all participating sites.

These approaches have several design limitations including: (i) single point of failure; (ii) lack of scalability; (iii) high network communication cost for global scheduler (i.e., network bottleneck, congestion); and (iv) computational power required to serve a large number of resource look-up and update queries on the machine running the information services.

Each participating site in the Grid needs to send an update of its most recent resource information to the meta-scheduler at constant periodic time intervals. Such frequent and massive information flow incurs high network communication at the meta-scheduler level. A performance study (Zhang, Freschl, and Schopf, 2003) has shown that existing information systems, which include MDS-2,3,4 (Czajkowski et al., 2001), RGMA (Podhorszki and Kacsuk, 2003), and Hawkeye (Pan et al., 2007), fail to scale beyond 300 concurrent users (Ranjan et al., 2007). With a large number of users and resources, a centralized or even a hierarchical scheduler that accommodates scheduling of a billion resource nodes for millions of users is simply not feasible.

3.2.2 Scheduling and Resource Allocation Delay

The resource allocation delay represents the amount of delay from when a job is submitted to a resource until it begins execution. Current resource selection approaches in a resource sharing environment (i.e., Grid environment) seek to reduce the total turnaround time (or more simply, some aspect of the total turnaround time such as execution delay or communication cost) (Hauswirth and Schmidt, 2005). Recent study has shown that the waiting times for resources under the EGEE environment can vary greatly, from a minute to one hour (Lingrand, Glatard, and Montagnat, 2009). For a job that requests a large number of nodes (e.g., more than 256 nodes), the waiting time is significantly longer, which can be up to a few hours[1]. This variability is known to significantly impact application performances and thus can be lead to major setbacks for time-critical applications that need rapid access to resources to meet their QoS (i.e., deadline) requirements.

Several scheduling algorithms have been proposed to resolve the unbounded waiting delay issue for parallel job scheduling (Ramamritham, Stankovic, and Shiah, 1990; Smith, Foster, and Taylor, 2000; Islam et al., 2003). The most frequently cited solution is advance reservation (AR) (Elmroth and Tordsson, 2009; Engelbrecht and Benkner, 2009). Advance reservation has been introduced in Grid environments to provide time Quality of Service requirements (QoS) for time-critical applications. It is incorporated as part of workflows (Cao et al., 2003) to provide the resource coordination in which the resource guarantee is achieved using a backfilling technique (Mu'alem and Feitelson, 2001). With advance reservation approaches, a user submits a job as a set of tasks with dependencies. These tasks are then reserved on a set of resource nodes prior to execution.

Although the advance reservation approach could ensure the QoS guarantee for the task execution, the advance reservation and backfill scheme requires the user to provide a runtime estimate for all submitted jobs (Feitelson and Rudolph, 1995; Park and Humphrey, 2008). This requirement introduces two problems. Firstly, observations have shown that the estimated-time provided by the user is not often accurate (Zhang et al., 2001). Unfortunately, even a slight amount of inaccuracy could affect job perfor-

[1] We observed that latency varies greatly with the chosen site - a job submission with a request for 256 nodes from EGEE Asian site can take up a few hours before the job is ready for actual execution.

mance considerably (F.Heine et al., 2005). Secondly, it is not often possible to predict the job runtime duration for some applications. For example, the runtimes for adaptive and interactive jobs are unpredictable since they change frequently during execution due to feedback from user actions (Liu, Nazir, and Sorenson, 2007).

3.2.3 Limited QoS Support

In a conventional Grid system, a meta-scheduler makes scheduling decisions based on the current state of resource availability information from all participating sites, but the scheduling decisions do not take into account a job's QoS parameters. As a result, the conventional meta-scheduling approaches tend to penalize large jobs that request a high number of resource nodes even if such jobs only run for short durations. For example, small jobs that request a few resource nodes can be accommodated easily while jobs that request a large amount of resource nodes are being forced to wait in the queue for a long period of time. Even at the job scheduler level, distributed resource managers (DRMs) such as Legion (Grimshaw, Wulf, and Team, 1997), Condor (Thain, Tannenbaum, and Livny, 2005), PBS (Henderson, 1995) and SGE (Gentzsch, 2001a) only provide limited support for the QoS-driven requirement. The DRM either focuses on minimizing the response time (sum of queue time and actual execution time) or maximizing overall resource utilization of the cluster pool. This limitation is a major setback for distributed applications which have QoS requirements (i.e., time-critical/deadline-driven jobs) since such jobs cannot afford to wait in a job queue for resources to become available. Therefore, without QoS-based scheduling in place, a meta-scheduler or a DRM cannot differentiate the level of priorities required for different applications because they assume that all job requests are equally important and thus neglect the actual levels of service required by different users. As a result, the meta-scheduler or the DRM may unnecessarily assign resource nodes to a best-effort job while an urgent job (i.e., one that is deadline-driven) is forced to wait in the queue.

Several works on economic-based distributed resource management and scheduling have tried to provide solutions to this problem (Chun and Culler, 2000; Islam et al., 2003; Irwin et al., 2005; Lee and Snavely, 2007). They proposed the concept of a virtual currency as a mechanism to differentiate between urgent and non-urgent jobs. While these studies provide a first step towards resolving the problem, there are still outstanding issues that are yet to be explored but that are important to address in order to realize a large-scale resource sharing system in practice. For example, at the moment, conventional resource sharing systems (i.e., Grid systems) only support fairly simple job models of a sequential job processing type (Yeo and Buyya, 2006). Advanced jobs that have QoS constraints (e.g., urgent (Beckman et al., 2006) and interactive steering applications) are not supported. Such applications have unpredictable and dynamic requirements during their execution because they need to meet peak demand or sudden surges in processing demand at application runtime. Therefore, a mechanism is needed to support such QoS requirements constantly during application execution.

3.2.4 Non-Cost Aware

The success of managing distributed resources on a national scale does not lie only in minimizing waiting and response times. Regardless of the underlying platform (i.e., clusters, supercomputers, desktops), users want to be able to execute their applications without making a huge investment in new hardware. Resource sharing is about leveraging the available resources and idle processor cycles to solve a problem more quickly while at the same time maximizing efficiency and reducing the total cost of ownership.

To justify the total cost of operating resources, the resource management system needs to consider the revenue that the system will gain by serving the applications' QoS requirements against the cost to operate resource nodes for satisfying such QoS requirements. However, adoption of distributed computing platforms and services by the scientific community is in its infancy because the performance and monetary cost-benefits for scientific applications are not perfectly clear. This is especially true for conventional Cluster and Grid systems which have been widely deployed for the purposes for providing data and compute resources. For example, the University of Oxford has invested in creating a campus-wide Grid that connects all large-scale computational resources and provides a uniform access method to 'external' resources such as the National Grid Service (NGS) and the Oxford Supercomputing Centre (OSC) (Wallom and Trefethen, 2006). However, it can be argued that a truly volunteer-based model such as the campus Grid system may not be sustainable without a cost-sharing scheme in which the hardware and operating maintenance costs are shared among many users. Therefore, an economic model (based on costing framework) similar to Cloud model (e.g., Amazon EC2 (Amazon, 2009)) is advocated whereby there is a virtual economy and an underlying costing model. Although there already exists a costing (economic) model in Cloud systems, further work is needed to bridge Cluster and Grid resources because there is currently a very weak costing model that is being used to connect the two. Incorporating a standard costing model is advocated because this would offer the possibility to integrate Cluster, Grid and Cloud resources seamlessly.

In this respect, there is a strong need for the resource management system to quantitatively measure the cost-benefit from serving the jobs for the users and the resource cost involved from operating the computing resources for running such jobs. In order to achieve this, a costing model needs to be formulated to capture the following questions: (i) what are the performance trade-offs in operating computational resources to serve jobs? (ii) What are the monetary costs of operating computational resources specifically for serving application QoS requirements? (iii) In light of those monetary and performance cost-benefits, how can the system ensure that jobs are served in the most cost-effective manner without incurring unnecessary overheads?

An economic-based approach has been advocated (Buyya, 2002; Irwin, Grit, and Chase, 2004) whereby applications are able to express the monetary value of their jobs as the price they will pay to have them run. In essence, applications/users are charged appropriately for the job execution service offered by the resource management system. Similarly, the system must also take into account the cost related to operating the computing resources (i.e., the resource cost). These include administrative and operating costs, such as purchasing servers, floor space, personnel, as well as the electricity cost to operate resource machines and cool them. Moreover, the system must ensure that the resource-related cost to run the jobs does not exceed the revenues earned from users/applications in order to justify the performance and monetary cost-benefits.

3.2.5 Lack of Service Level Agreements

More recently, a significant area of research has related to establishing agreements on the quality of a service between a service provider and a service consumer (Liu, Squillante, and Wolf, 2001; Barmouta and Buyya, 2002; John et al., 2003; Zhang and Ardagna, 2004; Yeo and Buyya, 2005). Service Level Agreements can be based on general agreements, e.g., framework agreements, that govern the relationship between parties and which also may include legal aspects and set boundaries for SLAs.

While there have been several recent research works on service level agreements (SLAs) in the Grid environment (Buyya, Abramson, and Giddy, 2000; Barmouta and Buyya, 2002; Buyya, 2002), as of today, there is no practical framework nor policies that support charging, scheduling and resource provisioning for job execution under a controlled environment where both the users and resource owners are operating on different cost models and options. Previous works (Liu, Squillante, and Wolf, 2001; Irwin, Grit, and Chase, 2004; Zhang and Ardagna, 2004; Yeo and Buyya, 2007) have considered economic-based scheduling, but only from the user's perspective. These works have considered job budget or job price as the main criteria to evaluate user performance. However, the job budget or price is assumed to be fixed regardless of how long the job runs. This is not a valid assumption since the longer the job runs, the higher the cost to serve the job.

Furthermore, prior works assume that the service level agreements between two parties are often established at the initial stage of application execution, but there is a lack of an option for users to modify the initial SLA agreements once they have been submitted. The lack of such an option prevents the system from supporting adaptive and interactive applications with varying QoS requirements (i.e., varying deadlines). Finally, prior works do not examine SLA-based policies that attempt to counterbalance the cost of satisfying application QoS requirement against the real operational cost for renting and using resource nodes.

Given the above limitations in prior works, a more comprehensive SLA framework is therefore needed to support a wide range of applications with different QoS requirements. Moreover, such a SLA framework should consider the different cost options offered by the resource providers, for example, the resource cost variability from the selection of available resources.

3.3 Requirements

In the previous section, we have identified several major shortcomings of current resource management infrastructures. Building on the strengths and weaknesses of previous works, we will now outline how we intend to address these above mentioned issues. We present our approaches and the fundamental requirements that we envisage a resource management system must meet in operating a large-scale distributed resource sharing system.

3.3.1 Resource Renting

In conventional distributed and Grid systems, a meta-scheduler or a broker is relied upon to control all resources worldwide and make all decisions on resource allocation and scheduling of all received jobs. Faced with a potential large number of sites, the central scheduling point represents a potential bottleneck for requests needing fast response. Therefore, there is a strong need for the removal of a centralized meta-scheduler to avoid the creation of bottlenecks and reduce the fine-grained management of requests and data transfers.

There is a need for a decentralized scheduling environment which deviates from the conventional centralized meta-scheduler paradigm that is often adopted in conventional Grid systems. We consider a solution whereby a limited number of 'Virtual Authorities' (VAs) are formed to their own customized ideal set of resource machines. A VA is a collection of resources controlled, but not owned, by a group of users or an authority representing a group of users. The owners of a resource recognize only the VA. All permissions, charging and blame for security breaches are associated with the VA.

Resource planning is the responsibility of the VA. Resources are rented on short-term and long-term bases to satisfy the application QoS requirements of their users. The VA acts like a retailer and trades existing stock with users in response to their requirements. It also acts as a stock taker and constantly monitors its stock of resources. If it feels it has too many in store, it will unload them and if there are too few, it will negotiate with resource providers for more resources. The VA is able to change its resource pool based on the goal of getting better deals. For example, during the daytime in England, a VA may find it beneficial to unload popular and 'expensive' local resources and replace them with cheaper resources from the night-time countries. The impact of the overhead of processing delays incurred should be minimal as the system scales both upward and downward dynamically.

A VA defines a local resource service that controls and manages local scheduling policy. It provides a localized exclusive environment for applications or job execution services using virtual resources created from rented hardware, which is supported by a pool of resources that it may either purchase or rent. The rental arrangement is possible because it makes agreements with other pool owners (resource providers) to rent some of their resources in times of high demand in exchange for rental fees.

Resource nodes are rented based on their types and rental period. A node is supplied with a set-up that enables the VA to select a specific operating system with the required software environment for the application. The recent progress in Cloud computing (Sullivan, 2009) has also made it possible for the resource provider to rent resource nodes using state-of-the-art virtual machine (VM) technology, allowing the VA to install an operating system and customize the software environment as required on a physical hardware machine. Since resource nodes can be added and removed dynamically from a system, the VA can anticipate resources of different types with minimal infrastructure and administrative cost and overhead. Such flexibility enables distributed and parallel applications to harness the combined power of geographically distributed resources. Moreover, as far as scheduling is concerned, the VA is independent from competitors and can instead concentrate on managing its own resources. This will result in a much reduced problem, and consequently, simpler and faster scheduling.

3.3.2 Costing Model

A desirable resource management system must support QoS-based resource allocation to serve a wide range of distributed applications which include parallel and interactive applications with varying QoS requirements. However, current resource management systems are not suitable for running applications with QoS requirements for several reasons.

First, most systems only allow users to express a single QoS requirement of their application in a job description. This is inadequate because an execution deadline is not normally static because it may change dynamically at runtime in response to execution complexity and/or user actions. An application may generate many tasks during execution with different execution sizes and each task may have a different deadline value. Second, even if the system allows the inclusion of multiple deadline values, this added feature can be misused by users. For example, users can deliberately specify very strict deadlines for all their jobs. This will have an undesirable effect on the performance because the system will not stop users from claiming that their applications need the highest quality of service.

To resolve these issues, the concept of virtual currency (Rappa, 2004) has been proposed: by providing a limited funding (i.e., monetary) capacity to users for running their jobs, users would be prevented from assigning strict deadline values to every job. The basic idea is to incorporate economic mechanisms into the resource allocation system. For example, the deadline value can also be expressed in terms of virtual currency (Chun and Culler, 2002b), in terms of 'monetary value' rather than a cash currency (e.g., pounds, dollars etc.). The advantage of the conversion of deadline to virtual currency is that it discourages free-riding and gaming by strategic users. Such an approach promotes fairness because applications that claim a higher priority for a job will have to pay more.

It is also envisaged that the currency must have the property for *self-recharging* (Irwin et al., 2005) whereby the spent monetary values are restored to the application's fund after some delay. The key idea is to recycle currency through the economy automatically while bounding the rate of spending by users. Currency budgets may be distributed among users according to any global policy; consumers spend their budgets to schedule their resource usage through time, but cannot hoard their currency or starve.

The monetary value assigned to each application is allocated based on its default priority (e.g., interactive steering applications generally will be allocated higher value when compared to best-effort applications). This initial value is expressed in terms of the accumulated total monetary value (TMV) which is to be recharged automatically, ensuring a stable fund for job execution but also preventing abuse and misuse by the applications. However, the amount of TMV assigned must be capped at a maximum of a certain value (i.e., maximum value), regardless of the level of application priority. This is to ensure that the lowest priority application also gets a fair chance to be scheduled.

The advantage of the conversion of time to money is that it allows the mechanism for the applications to specify how important a job is in comparison to other jobs. For example, consider two jobs: Job A and Job B. If job A is twice as important as Job B, then Job A may be assigned two times higher monetary value in comparison to Job B. Such an approach promotes fairness because applications that claim a higher priority will have to pay more. Most importantly, this gives incentive for the application to specify the true deadline value for each job since the amount of TMV assigned to each application is capped for the whole period of its execution.

Employing virtual currency would resolve the issue of differentiating high priority and low priority jobs. A costing model provides the mechanism to capture revenues generated from applications and also expenses incurred from operating resources. From the user's perspective, the management system must aim to minimize the penalty cost. The penalty cost is incurred when the system fails to meet the application's QoS requirements (i.e., deadline). On the other hand, from the resource provider's perspective, the system's objective is to minimize the rental cost. The rental cost derives from the need to deploy resource nodes from external providers and the operational costs to operate them. For example, the rental cost includes the administrative cost (i.e., deployment cost) and the operational cost (i.e., electricity, personnel, floor space, etc.).

If the only objective is to minimize the rental cost, the system could rent the minimum amount of nodes. However, it is also equally important to reduce the penalty cost. The costing model should capture a performance metric that minimizes both the penalty cost and the rental cost. Drawing from these observations, we can combine and capture the revenues, penalty, and expenses with a single profit metric. The advantage of the profit metric is that it represents a single value that captures overall trade-offs between satisfying the application QoS requirements versus the cost of renting resource nodes. It addresses the trade-off between the cost of rental and the lost opportunity if customer demand is not met. Therefore, it is envisaged that the profit metric offers a clear measure for the system to evaluate its rental decision. In essence, this provides a valuable tool for capacity planning, and it provides the foundation for improving productivity, managing costs, and return on investment for renting resources.

It is also important to consider resource cost from the perspective of resource providers. For example, what should the resource provider charge for renting out its resource nodes? In particular, are some cost models better in the sense of allowing lower rental fees for long-term rentals compared to short-term rentals? Although both users and resource providers act independently, they do not operate in a vacuum. Users will aim to maximize their quality of services, while resource providers will aim to increase their machine utilizations as much as possible. With conflicting objectives of the users and the resource providers, how can we ensure that both metrics can be satisfactorily met? A costing model must therefore consider both sides of the issue in order to serve the interests of both users and resource providers.

3.3.3 Service Level Agreements (SLA)

A job description defines the resource requirements that the meta-scheduler should refer to when determining the resource nodes to allocate. The job description typically includes the minimum resource requirements needed to run a job (e.g., number of CPUs, number of nodes, memory, software libraries, etc.). Based on these requirements, the meta-scheduler delegates the job to the most appropriate and suitable distributed resource manager. Hence, a job description only captures the job requirement prior to application execution.

However, in practice, the resource requirements for jobs frequently change during execution because of the need to meet peak demand or sudden surges in processing demand at runtime. As a result, a static job requirement alone does not often reveal sufficient information for the system's scheduler to successfully meet the QoS requirements of the application throughout its execution.

To resolve the issue, it is envisaged that the scheduling decision should not only take account of an individual job's information, but the system's scheduler should also consider the knowledge of the anticipated future workload. Therefore, an additional control is needed so that the system has information pertaining to the overall execution behaviour from the start of user submission until the end of user submission.

It is anticipated that this control can be provided by means of service level agreements (SLAs), or long-term contracts. Unlike an individual job description, such SLA contracts include additional information such as the total load that the application can impose, the application total offered monetary value, and its total running time. Such contracts may also include penalties for poor performance: agreement on how much the system should be penalized for not meeting specific QoS requirements e.g., if the response time is too long for many jobs or if too many deadlines are missed.

Given such a control, the system can optimize rental decisions in the long term and plan ahead accordingly. Short-term rentals can be employed to accommodate sudden demand from running applications, while the VA can also choose to rent long-term capacity for accommodating near-future demand depending on the information provided by the SLA. For instance, if the application demand is predicted to be at a stable rate over a period of time, the VA can plan to rent less expensive (cheaper) resource nodes for a long period.

3.4 VA Framework

In this section, we give an overview of our solution and describe some of the components and the challenges of building such a resource management system. We envisage that our proposal for a cost-effective resource management that can realize the vision of a national and large-scale resource sharing system supporting millions of users with a wide of range of processing requirements and with potentially billions of heterogeneous resources.

3.4.1 Multi-tier Framework

The VA introduces the possibility of managing geographically distributed resources in significantly 'smaller' computing units than those that are currently managed by a global Grid system. There is no optimal size for such 'smaller' units as this is mainly guided by a complex symbiosis between usage pattern and timing.

The most distinctive feature of a VA is that it decouples the fundamental rental mechanism from scheduling and resource allocation policies. In order to achieve this, a VA has several distinctive features that differ from the conventional meta-scheduler paradigm. Figure 3.1 illustrates the high-level relationship between various components of the system. As observed, a VA consists of three main components: Application Agent (AA), VA Scheduler and VA Negotiator.

The VA is built on a multi-tier paradigm that differentiates the roles of application management, scheduling and resource provisioning. The application management is handled at the upper tier. The main operation of the upper-tier is to translate the application processing requirement into explicit resource requirements. The upper-tier interacts directly with the middle tier via application programming interface (API) calls.

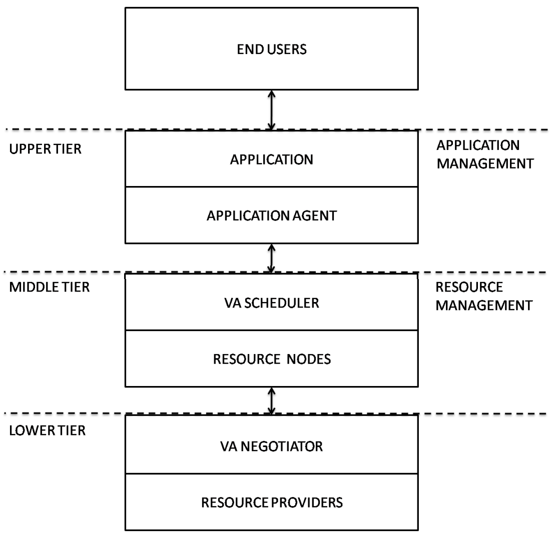


Figure 3.1: The interaction relationship between users, applications, application agents (AAs), VA Scheduler, VA Negotiator, resource providers, and resource nodes.

The VA Scheduler is handled at the middle tier. Upon receiving a job request from the upper tier, the VA Scheduler places the job in the waiting queue and schedules the job appropriately in accordance with the QoS requirement to resource node(s).

The resource level is observed by the VA Negotiator at the lower tier. At a regular time interval or when a specific event has occurred, the VA Negotiator reviews the current resource availability status and/or the estimation of projected future workload, and then it rents the amount of resource nodes required from resource providers from multiple resource providers based on the rental policy.

The multi-tier approach essentially differentiates the roles of application management, scheduling and resource provisioning. Each tier is independent of another and therefore the AA, VA Scheduler and VA Negotiator can perform their own tasks without any conflicting objectives.

3.4.2 Principle of Resource Renting

The provision of sophisticated rental policies is therefore essential for the economic viability of a VA. The main purpose of renting is to acquire external/remote resources for local scheduling usage, in order to complement supply demand of local resources. A rental policy must provide a set of rules to determine the type and the amount of nodes to rent, and when to rent them. The responsibility of a VA is then to offer a cost-competitive service in order to attract users.

The concept of resource renting is shown diagrammatically in Figure 3.2. Renting in the VA involves the exchange of resource nodes from the resource provider in return for a rental fee. Resource renting is not necessarily a one-time activity that occurs at the start of application execution. In the standard job submission process, the resource nodes are assigned to users only when the job is first scheduled prior to execution. The two drawbacks of such a standard job submission approach is the likelihood of fragmentation and long waiting delay (Feitelson and Rudolph, 1995). Fragmentation occurs when the available (free) nodes are left idle, but the allocation policy fails to allocate these nodes to the waiting jobs. The waiting delay occurs when submitted jobs are forced to wait in the queue for a long time until

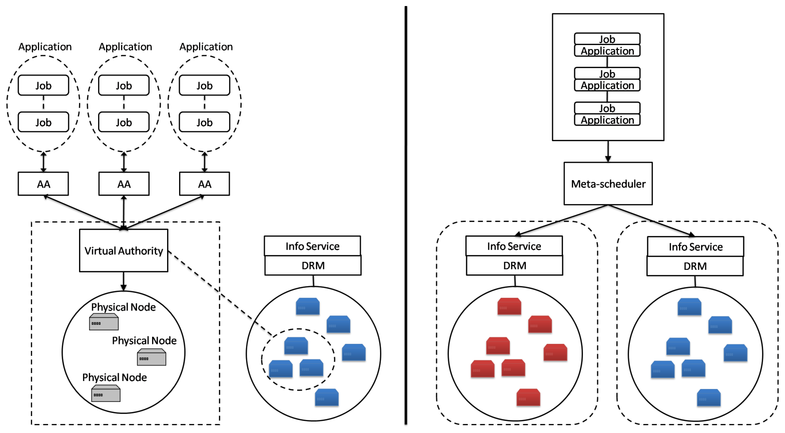


Figure 3.2: Resource Renting vs. Meta-scheduling.

the requested nodes become available. The fragmentation and long waiting delay issues can be resolved by allowing the application to interact with the VA Scheduler (middle tier) at runtime. The application can request additional nodes in response to load changes in application. As a result, the VA Negotiator rents the additional resource nodes necessary on demand from the resource providers worldwide. Consequently, the resource pool within the VA is constantly changing as the system scales in size both upward and downward.

In essence, the VA is made up from ready-use resources for applications to use. The ready-use resource is monitored constantly by the VA Negotiator. Based on the anticipated demand, the VA Negotiator may react on a reactionary basis in response to sudden demand for resources and/or it may choose to evaluate its resource level periodically at regular intervals. For example, if the application workload is predicted to be stable over a time period, the VA Negotiator may rent the minimum amount of nodes in intervals until there is a sudden demand for further additional nodes. Alternatively, a combination of long-term and short-term rentals may be advocated in a situation where the resource demand is stable throughout application execution with only few sudden bursts. Therefore, the choices of rental options (i.e., rental policies) can have considerable effects on the VA's performance.

3.4.3 Application Management

The application management constitutes the upper tier of a VA system. We briefly identify the requirements for three core components within this tier: end users, application and application agent (AA). The AA provides the mechanism for applications to request additional resources at runtime (e.g., urgent and interactive steering applications).

3.4.3.1 End Users

End users are only interested in data output and the results of application executions. They do not care about the physical location and configuration of the system that delivers the services as long the required quality-of-service can be met. If a user wishes to execute an application, he/she should instead start the agent for that application. The application agent (AA) will in turn negotiate with the VA for resources on behalf of the application.

3.4.3.2 Applications

Current resource sharing systems are known to operate on a best-effort basis, sharing the resource nodes among its users with equal priority. Best-effort systems are characterized by a lack of service level guarantees, and the quality of service they provide may vary over time due to workloads, resource level, etc. Applications are also restricted to dumb resource management – the user has to manually specify the number of requested nodes prior to job submission (Liu, Nazir, and Sørensen, 2009). There is no collaboration with the underlying schedulers at runtime. However, for most distributed and parallel applications with QoS requirements, there is a strong need to balance the computational load over resources in a dynamic manner when the execution environment changes. This requires an active collaboration from both the application and the VA during application execution (at runtime) for requesting additional nodes and/or for releasing its current allocated resource nodes. In achieving this, the application must communicate with its agent by submitting multiple job requests at any point of its execution in order to meet sudden surges in processing demand. This execution model differs from the conventional job submission where there is only one job submission per application.

3.4.3.3 Application Agent – Upper Tier

The AA provides a layer between the user and the application. A user is typically unable to provide the information required by the job scheduler. However, it is difficult to perform resource management without at least some knowledge of the basic parameters involved and some understanding of what the resource can deliver. The AA acts as a 'semi-intelligent' user in such situations. Each application is associated with one AA.

The rules enforced by the AA are partly controlled by the application developers. The application developers may embed some knowledge of processing demand. For example, they may embed prior knowledge of the fraction of resources which are needed to perform specific job and tasks; this may include a priori knowledge of processing times for specific algorithms of computations etc. Using the rules embedded by the developers, the AA constantly monitors application execution and reacts appropriately according to these rules.

The AA provides the following important functionalities: (i) it acquires and releases resource nodes on behalf of the application during execution; (ii) it translates the application processing requirement into resource requirements such as the requested number of nodes and quality-of-service (QoS) parameters (i.e., deadline value); and (iii) it deploys proxy services to enable transparent execution on remote nodes irrespective of the physical location of the nodes and the location of the access points. In practice, the AA is invoked by the application via an application-specific library with simple Application Programming Interfaces (APIs). The application makes a direct request for resource nodes. Upon allocation, the AA is given exclusive right to the nodes within the allocation period[2].

For each job request, the AA should specify the following on behalf of its application: (i) the maximum monetary value of a job (this represents the fraction of monetary value *V* out of *TMV* ($\frac{V}{TMV}$) it is willing to spend) and (ii) job execution deadline D_j. Both the monetary value and the request deadline

[2]If multiple virtual machines like XEN or KVM run on each physical node, then a node here implies a virtual machine.

reflect the importance (priority) of a specific job to the application. Here, we assume a self-recharging currency model (Irwin et al., 2005) that assigns initial funding (i.e., monetary values) and replenishes the virtual currency at the end of the application completion. The application is issued a limited fund (in terms of TMV) by a trusted bank service (Barmouta and Buyya, 2002) at the initial stage of job execution. The initial monetary values are assigned by the bank service to the user upon the acceptance of SLA agreement (after negotiation). Upon SLA acceptance, the application may freely assign high/low values to its jobs within the limited funding capacity (i.e., $\sum_{j=1}^{N} V_j < TMV$). For example, within its pre-allocated total monetary value, the application may freely assign a higher monetary value for an urgent job request (e.g., real-time/interactive response). Similarly, it may assign a lower monetary value for a non-urgent request (e.g., best-effort response). Such an approach enables the application to submit multiple jobs at any point during execution while ensuring fairness among all other competing applications. This approach also enables the scheduler to satisfy the application's QoS requirements more efficiently.

3.4.4 Resource Management

3.4.4.1 VA Scheduler – Middle Tier

The VA Scheduler component is responsible for dynamically assigning applications' jobs/tasks to available rented resources. Rented nodes are shared within the VA to be used by several applications, and these applications are charged for using resource nodes. The main objective of the VA Scheduler is to efficiently schedule jobs so that waiting time and missed deadlines (penalty cost) can be minimized. In achieving this, the VA Scheduler performs scheduling based on the job parameters such as number of requested nodes, deadline, and monetary value. The VA Scheduler is invoked whenever a job arrives, a job completes, or when there are changes in available resources.

The VA scheduler carries out resource allocation based on a resource permit concept. A resource permit is simply a fine-grain permission that enables the AA to make direct contact with individual nodes via secure communication links. When a job is allocated with resource nodes, the VA does not send the job for execution but issues a resource permit instead. A resource permit contains information pertaining to the location of the nodes (i.e., public IP addresses, hostnames etc.) including the public key certificate for enabling direct access to individual resource nodes. As a general rule, only one permit is granted to a node at any one time. The permit approach ensures that the activities of the AA are isolated from other AAs that run within the same VA.

Unlike any other conventional scheduling systems, the VA Scheduler is responsible for managing the earning revenue from job execution. At the end of job completion, the VA Scheduler updates the revenue. The revenue is charged and gets updated once the job request has been assigned to resource node(s). At the end of job completion, the VA Scheduler determines whether there have been any QoS violations. If so, the AA is compensated with the penalty cost. Hence, it is important for the VA Scheduler to monitor the execution of all running jobs for any QoS violations at runtime.

3.4.4.2 VA Negotiator – Lower Tier

The VA Negotiator is responsible for renting nodes, releasing nodes, and managing rented nodes from resource providers based on the rental policies enforced. To accommodate the demands of various users and applications, the VA Negotiator reacts whenever a job arrives, a job completes, or at a regular time interval. When any of these events occur, it may rent additional nodes, or decide not to take any action at all. For example, the VA Negotiator may only choose to react when the resource level within the VA has reached a specific threshold.

Resource nodes are rented over some period of time in accordance with SLA agreements made between the VA Negotiator and resource providers. When the nodes are rented out, the resource provider loses usage of these nodes and therefore, is paid a rental fee. A rental fee is charged based on node capacity (i.e., processing speed) and rental duration. During the rental period, the VA Negotiator is in control of how it would like to use the nodes. Each node is bound by a rental agreement that gives access rights to the node for a certain duration.

The rental policies are categorized into two primary modes: reactionary mode (on demand) and proactive (long term). The reactionary (on demand) rental mode reacts whenever there is a sudden demand for additional nodes when a job arrives, when a job completes, or when the number of available resources changes. Alternatively, the proactive mode reacts when a certain threshold of specific-defined parameters has been reached e.g., load, resource level etc.

The rental policies specifically define a set of algorithms and rules to determine the type and amount of nodes to rent, and when to rent them. Examples of such policies include the aggressive and conservative rental policies which we will propose later in Chapter 6, which can be successfully applied in a VA setting. Examples of such policies are constructed from `rent()` and `release()` operations. The `rent()` operation rents additional nodes, whereas the `release()` operation releases nodes from the system. Both `rent()` and `release()` operations are invoked when a specific event has occurred or at periodic time intervals. Once the `rent()` operation is triggered, the system waits for nodes to arrive. Similarly, nodes are held by the system until the `release()` operation is invoked.

Another key aspect of the VA is the ability to discover resource providers without relying on a centralized global broker, and the ability to discover the most suitable resources at a reasonable cost in the shortest time. To eliminate potential performance and reliability bottlenecks, we argue that there is a need for decentralized resource discovery without global knowledge about the Grid condition. This is because decentralized resource discovery would ensure fast responses to resource renting requests. With this in mind, a number of research efforts have been undertaken that recognize the need for decentralized resource discovery solutions. For instance, Das and Grosu (2005b) explore a decentralized combinatorial auction that can assist the service provider to discover resources without relying on a global broker. Such an auction system performs resource discovery by choosing the bidder with the lowest cost criterion.

More recently, Iordache et al. (2007a); Fiscato, Costa, and Pierre (2008); Costa et al. (2009) have proposed peer-to-peer solutions that resolve the issues related to the discovery of available resources. Such solutions could potentially enable large-scale resource discovery to be carried out with only very partial knowledge about the structure of resource providers as a whole. Their works are complementary to our work, but it is beyond the scope of the thesis to delve deeper into the algorithms and solutions that can be used as a means of effective resource discovery. Instead, our contribution in this thesis is to propose cost-effective resource management, architectural designs and the necessary mechanisms that need to be in place for the realization of a large-scale resource sharing system.

3.5 Chapter Summary

In large-scale distributed environments, there are a large number of available resources and a large number of users who would dearly love to use them. Both the resources and the users are generally distributed across the globe and belong to a large number of institutions or resource providers. Some environments may give a limited number of people free access and others may give a much larger community access in return for financial recompense. The problem is how to manage these geographically distributed resources more efficiently for a wide range of distributed and parallel applications with varying QoS requirements.

In this chapter, we have identified a number of unresolved issues with current distributed computing infrastructures. In particular, we have focused on meta-scheduling issues in a distributed scheduling environment; particularly on scalability issues with a global meta-scheduler, resource allocation delays and other QoS aspects including the dynamic QoS requirements of running applications. We have also stressed the difficulty in sharing resources among different resource providers and the inability to specify user-level QoS fairly due to the lack of cost modelling and lack of SLAs in current distributed environments. To mitigate these shortcomings, we have argued that there is a strong need for a cost-based interaction (seemingly similar to supply-demand interaction) between the different resource providers and that this interaction should govern the scheduling process. This economic model comprises of renting resources from resource providers at a price and users being able to satisfy their QoS at a virtual cost when multiple users compete for conflicting QoS. Furthermore, a mechanism is needed to enable users to express and negotiate SLAs with resource providers so that trust relationships can be established between them to promote long-term rentals.

We have presented the high-level requirements that will need to be adhered to in order to resolve many of these issues. There are of course many areas that have not been specifically addressed in this chapter. For example, one may wish to address issues of security, trust, market dynamics of supply and demand, and resource discovery. However, for the most part, these issues are beyond the scope of our work, but they have been explored extensively in previous studies (Wolski et al., 2001b; Basney et al., 2005; Iordache et al., 2007a; Fiscato, Costa, and Pierre, 2008; Costa et al., 2009; Jabri and Matsuoka, 2010). For example, what should the provider charge for the resources in order to benefit from renting out its resources? The pricing scheme is not inconsequential because resource providers may behave strategically to keep their costs down and to maximize the return on their investment. Given those circumstances, how would market equilibrium be achieved and how would certain rental policies affect pricing?

Since our work focuses on the policy choices made by the VAs to keep their costs down and to maximise their return on investment, the effect of bidding strategies, pricing schemes and market dynamics imposed by resource providers are beyond the scope of this thesis. However, our work could provide a useful starting point for examining the impact of uncertainty in demand, resource availability and resource costs, the findings of which could serve as basis to further explore and evaluate market concepts such as user bidding strategies (Wolski et al., 2001c; Chun et al., 2005) and auction pricing mechanisms (Waldspurger et al., 1992; Lai et al., 2004; Das and Grosu, 2005b).

The primary requirements for the construction of cost-effective resource management are threefold and can be summarized as follows:

- Costing Model
  - The costing model takes advantage of the fact that different resources in disparate locations have varying usage levels. By creating smaller divisions of resources called 'Virtual Authorities' (VAs), organizations would be given the opportunity to choose between a variety of cost models, and each VA could rent resources from resource providers or potentially from other VAs when necessary, or rent out its own resources when it is underloaded. Furthermore, the responsibility of a VA is also to offer a cost-competitive service in order to attract users. The success of a cost-effective service would largely depend on two factors: (i) user responsiveness (high QoS satisfaction and lower penalty cost) and (ii) lower resource-related cost (i.e., rental cost). It is envisaged that information on job execution deadlines, job monetary values, and resource costs could be used effectively to maintain the economic viability of the VA. A costing model should enable the VA to quantitatively evaluate its conflicting objectives in order to minimize operating and rental-related costs subject to application satisfaction-level constraints.
- Resource Renting
  - Resource renting provides a mechanism for the VA to retain management authority and therefore there is no short-term dispute to resolve with the resource providers. Resource planning and renting are the responsibility of the VA. Its rental policy should provide a set of rules to decide what, when and how many nodes to rent from resource providers. The provision of a cost-aware rental policy that makes use of a costing model is essential for effective resource management.
- Service Level Agreements
  - The provision of an SLA mechanism that takes account of long-term contract information as well as the individual job is essential for effective rental decision making. Therefore, a mechanism for enabling users to negotiate long-term application/job execution contracts is advocated to increase the cost effectiveness of the VA.

The above requirements were motivated by our practical experience with and observations on the deployment and use of current distributed computing infrastructures for distributed and high performance computing projects over the last few years. This experience has enabled us to identify the difficulties in operating resources on a global scale and reconciling the interests of users, applications and the various resource providers from different administrative organizations. As a result, this has led to the foundation of the VA approach, which aims to address the unresolved issues related to resource management in current distributed computing infrastructures.

Chapter 4

HASEX Resource Management

In Chapter 3, we have proposed a Virtual Authority (VA) system to simplify the scheduling and resource management issues in distributed computing environments. A virtual authority acts as an intermediary that aggregates dispersed heterogeneous resources to serve individuals and/or a group of users. All permissions, billing and blame for security breaches are associated with the VA. A special characteristic of a VA system is the ability to 'rent' resources temporarily from external resource providers worldwide (e.g., clusters and Grid sites). Once rented, the nodes are reserved specifically to a requested VA for an agreed rental period.

In this chapter, we present the design and implementation of HASEX – a prototype system that partially realizes the ideas presented in chapter 3. HASEX is a rental-based resource management system that offers the capability to rent groups of heterogeneous resources across multiple resource providers. The key feature of HASEX is the ability to serve distributed applications with minimal infrastructure and operating costs. It is controlled by a rental policy which is supported by a pool of resource nodes that the system may rent from external resource providers. Unlike other resource management systems, HASEX differentiates the roles of application management, job scheduling, and resource renting. This approach significantly reduces the overhead of managing distributed resources.

This chapter is further organized as follows: we present the architectural design of the HASEX resource management system, detailing its key features, its architecture, its components and services. We then describe the implementation of HASEX. We present the results of our experimental evaluation and describe the advantages of the approach. We next present a discussion of related work and a critical appraisal of our approach, and finally we conclude the chapter.

4.1 HASEX Architecture Design

In this section, we present the architecture of HASEX. HASEX consists of a collection of higher level services, and it provides a number of useful features that aim to provide scalable, rapid, dynamic and flexible application execution. HASEX also provides a software framework which is composed of a library of interface routines (APIs) that enables an exclusive execution environment for the individual application.

4.1.1 Key Features

In this subsection, two key features for the HASEX are described. The key features of HASEX are that it provides (i) a resource management system with scalable renting capability over a selection of nodes from multiple resource providers and (ii) a cost-effective system that offers better resource availability, high utilization, performance, and scalability at lower cost, and the opportunity for incremental investment and immediate return.

Scalable Renting Capability. The main functionality of HASEX that differentiates it from other resource management systems is its ability to manage a set of heterogeneous resource nodes from multiple external resource providers and control them as if they were a single scheduling environment. Furthermore, unlike conventional Grid systems, HASEX rents resource nodes at the resource-level instead of at the job-level. To promote rapid deployment of nodes and scalability, HASEX decouples the scheduling and rental aspects of the system. The scheduling aspect is managed by the VA Scheduler, whereas the rental aspect is controlled by the VA Negotiator. The VA Scheduler makes decisions independently. The separation of responsibilities between the VA Scheduler and VA Negotiator enables the system to be more scalable because individual components can employ different policies to maximize their objectives.

Cost-Effective Resource Management. HASEX offers the capability to request resource nodes from external resource providers. It forms a rental agreement and releases the nodes when their rental period has expired. Once a resource provider has agreed to rent a node, HASEX takes control of the node during the rental period, and the charging and billing process takes place. Based on this renting model, HASEX offers cost-effective resource management that provides high application quality-of-service (QoS) satisfaction with minimal infrastructure and operating cost.

4.2 HASEX Core Services

HASEX supports parallel and distributed applications by providing transparent remote execution, which allows running jobs or tasks to be executed and controlled as if they were running locally. It also provides the mechanism to manage multiple jobs for the whole period of application execution.

HASEX supports two main modes for job submission: conventional job submission and the Application Agent (AA). First, users can submit jobs through a conventional batch system (e.g., SGE or PBS). Second, HASEX also provides an application-specific library via the AA that allows adaptive reconfigurable parallel/distributed applications to request additional nodes dynamically at runtime with very minor modifications to the original application code. The AA interface was built on previous work (Liu, Nazir, and Sørensen, 2009) that allows adaptive and reconfigurable applications to request additional resource nodes at runtime and releases them when they are not needed. Adaptive and reconfigurable applications (Islam et al., 2003; Agrawal et al., 2006; Park and Humphrey, 2008) refer to the distributed or parallel applications that are able to adapt to the various dynamics of the underlying resources. This implies the ability for the application to modify its structure and/or modify the mapping between computing resources and an application's components while the application continues to operate with minimal disruption to its execution.

Each application is associated with an AA. The AA simply provides an API (Application Programming Interface) that allows the application developers to insert request calls within the application code. The idea is to allow the developer to specify the condition when the application should seek additional nodes to accomplish a specific task. The AA makes use of this information to determine whether more or less resources are needed throughout different periods of application execution. Therefore, no modifications to the original application code are needed apart from a few API calls that need to be inserted in the application code to link the communication between the application and HASEX. To the best of our knowledge, this special ability is currently lacking in current resource management systems.

In this section, we present a high-level overview of the HASEX resource management system. Figure 4.1 presents a layered view of its subcomponents. The framework is built upon a number of services which we will describe in detail in the subsequent sections.

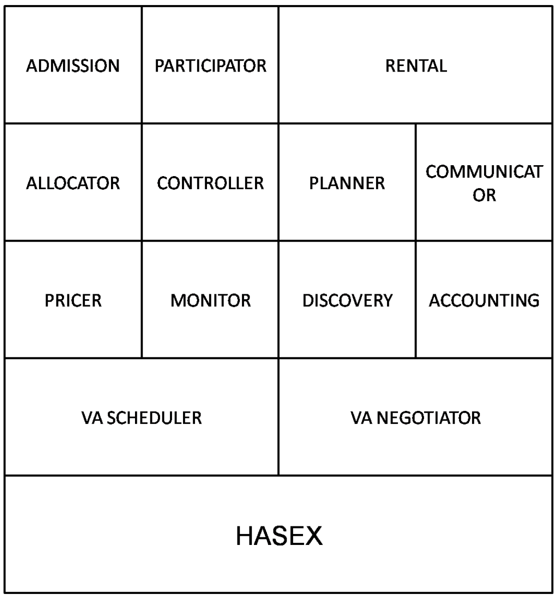


Figure 4.1: Core services of HASEX.

4.2.1 Services Interaction

The HASEX system has two core components: the VA Scheduler and the VA Negotiator. The VA Scheduler performs job scheduling and resource allocation while VA Negotiator performs node discovery, node negotiation, and manages rental aspects. The VA Scheduler and the VA Negotiator components have their own respective services and these services interact and exchange information with one another to accomplish a task.

In this section, we summarize the overall relationships of the services of the VA Scheduler and the VA Negotiator components with the application, the AA, and resource providers. The service interaction between the VA Scheduler and VA Negotiator are depicted in Figure 4.2. The interaction links between services represent how services communicate with one another. These services are independent and modular in that they can be enhanced or replaced without affecting other services.

Firstly, we can see that the AA component consists of three basic core services: Local Communi-

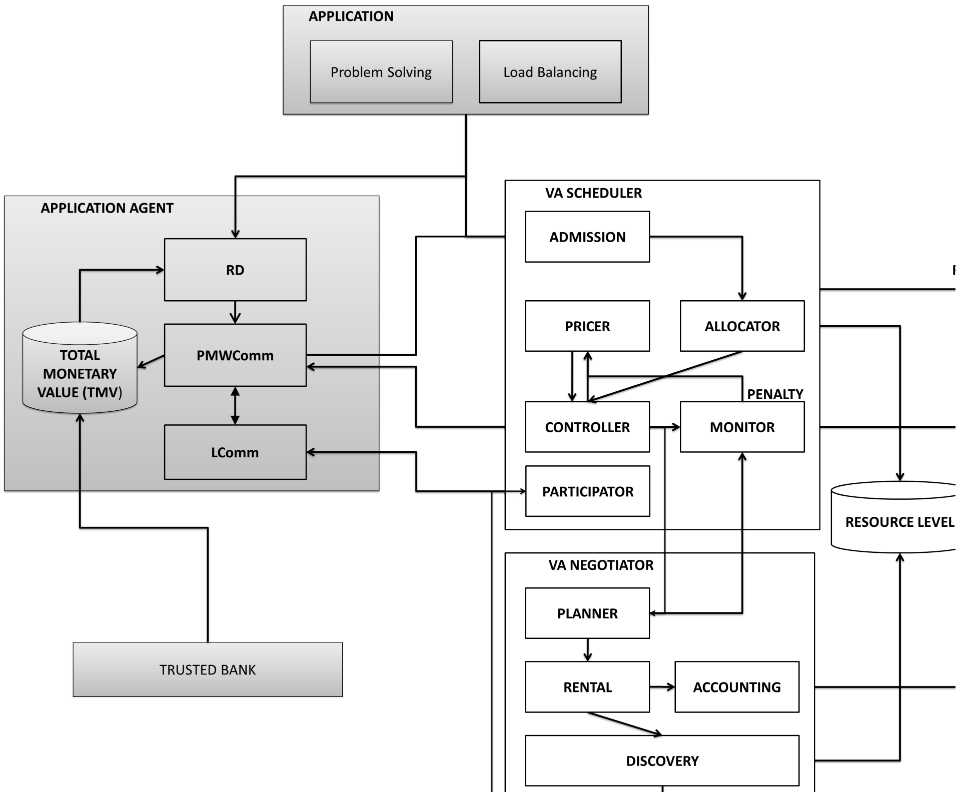


Figure 4.2: Interaction between AA, VA Scheduler, and VA Negotiator components.

cation (`LComm`), Process Management Communication (`PMWComm`), and Resource Discoverer (RD). As stated earlier, each AA is associated with a single application, although many end users can use the same application simultaneously to submit jobs. A job request is normally triggered by the application when there is a sudden demand for additional nodes due to an increase in the processing requirement resulting from complex calculations or user actions. When a job request is triggered, the AA's RD translates the processing demand into an explicit job request. The job request includes resource requirement parameters such as the number of requested nodes and the job deadline value (i.e., deadlines by which users want their jobs to be executed).

Based on the information regarding the job deadline value, the `RD` assigns a monetary value. The monetary value is a fraction of the Total Monetary Value (TMV) which is obtained from a third party i.e., the trusted bank (Barmouta and Buyya, 2002). The trusted bank is an external part of the HASEX system and we assume that it provides the mechanism for the application to purchase the amount of credits (i.e., monetary values) they need in order to execute their jobs/tasks on the VA. When the job request (i.e., resource requirement) has been prepared, the `PMWComm` then sends the job request to the VA Scheduler and waits for a response.

Secondly, the VA scheduler consists of six major services: `Admission`, `Pricer`, `Allocator`, `Controller`, `Monitor`, and `Participator`. The `Admission` service first authenticates the job or the AA's request. The `Admission` verifies whether the job contains the appropriate information (i.e., number of requested nodes, deadline etc.) before it is accepted for scheduling. If it is accepted, the job is placed in the queue, and `Allocator` performs the appropriate scheduling and resource allocation. When a job is allocated to a node, the `Allocator` also updates the resource availability to the RESOURCE LEVEL repository. If the job request cannot be accommodated due to insufficient nodes, the shortage of resource level is notified to the `Controller`. The VA Negotiator is then made aware of this situation via the interaction between the `Controller` and the `Planner` services. If there are available free nodes to serve the job, the `Allocator` issues a reservation request for resources from the distributed resource manager (DRM). Reserved resources are then returned for job execution.

If the request was originally made by the AA, the `Allocator` issues a resource permit containing the nodes' information instead. Given this resource permit, the AA's `PMWComm` spawns a `LComm` for each allocated node. The `PMWComm` then establishes a one-to-one communication session with the individual resource node. The `PMWComm` then performs the necessary data input/out transfer and monitors its own respective node for each task execution.

At the end of each job completion, the `Pricer` is triggered to record the monetary value earned from each job/task completion. In the case of a QoS violation, the `Pricer` updates the COST repository to compensate the penalty cost. The `LComm` also sends a notification message to the `PMWComm` upon job completion. Upon this notification, the `PMWComm` collects the execution results for the application.

We now describe the functionalities of the VA Negotiator services. The VA Negotiator consists of five core services: `Planner`, `Rental`, `Accounting`, `Discovery` and `Communicator` services. The `Planner` service implements the rental policy that reacts in accordance to certain rules. For example, it may respond by renting additional nodes when a specific condition has been triggered (e.g., when the resource level within the VA has reached a specific number). The `Rental` service issues asynchronous requests so that arrival of new nodes does not interfere with scheduling functionalities. This decouples scheduling and rental activities.

Upon arrival of new nodes, nodes are registered to the RESOURCE LEVEL repository. The `Accounting` service keeps track and monitors the resource cost that is associated in renting a node (i.e., infrastructure and operating cost). The `Accounting` service then records the expense to the COST repository.

In summary, a VA Scheduler performs all the related scheduling resource allocation and charging activities, whereas the VA Negotiator is responsible for handling all renting activities and maintaining a record of the expenses incurred from these renting activities.

The VA Scheduler accepts two modes of job submission: conventional job submission or the AA. The conventional job submission allows the user to submit and monitor jobs sent to the DRM. Alternatively, the AA submission approach makes it possible for the application to have direct and secure access to resource nodes for supporting interactive sessions. Regardless of the mode of submissions, the VA Negotiator handles all the infrastructure and operating costs (i.e., resource costs) associated with rental activities[1].

[1]The cost model will be described in more detail in Chapter 6.

The separation of interests between the VA Scheduler and the VA Negotiator promotes scalability and provides rapid access to resources. Although both the VA Scheduler and the VA Negotiator have conflicting performance objectives, they do not operate in a vacuum. Each component has a number of services that support them, and as we can see from the interaction between them (as shown in Figure 4.2), there is a clear case for collaborative relationships.

4.2.2 Job Submission

HASEX supports a wide range of high performance computing applications. In the subsequent subsections, we briefly describe how distributed and parallel applications are supported by HASEX through two job submission models.

4.2.2.1 Standard Job Submission

Users submit batch jobs either through the portal, workflow manager or the application by starting the application agent. Through the portal, the user will be given the option to submit a collection of batch jobs, cancel a collection of batch jobs, and also obtain status information on running batch jobs.

4.2.2.2 Application Programming Interface

There is an emerging area in the scientific applications field, urgent and on-demand High Performance Computing (urgent HPC), which requires a great capacity of computing resources at a given time (Islam et al., 2003; Park and Humphrey, 2008; Cencerrado, Senar, and Cortés, 2009). Examples of such applications include fluid dynamics (Nivet, 2006), financial analysis (Hu, Zhong, and Shi, 2006), molecular dynamics (Kobler et al., 2007), structural analysis (Alonso et al., 2007), to name a few. Such applications are typically adaptive or interactive applications since they may spawn multiple jobs during execution and the jobs may constantly pass parameters between themselves at runtime. In such a scenario, we consider the application as adaptive because it allows significant changes to its internal dataset structures during execution. Such applications are also interactive because of the need to support human interaction. Effectively, they are able to reconfigure themselves, self-optimize, and migrate to adapt to the characteristics of the underlying execution environments. They have the following properties: they (1) make use of new computing resources during execution; (2) perform internal load balancing; and (3) are resilient to resource failures.

HASEX offers the application programming interface (API) that supports such applications via the application agent (AA). The AA was originally built in our earlier work (Liu, Nazir, and Sørensen, 2009) to support adaptive applications in distributed and parallel computing. The AA offers a set of core services that allow reconfigurable applications to request nodes or to relinquish nodes during execution. The AA is a software framework which is composed of a library of AA interface routines (API) that replaces the need for user-provided execution information prior to job submission with a transparent software-controlled process. In essence, it hides the complexity of selecting the right resources to ensure execution performance metrics. It also provides the flexibility for applications to request nodes on demand at runtime in contrast to a traditional batch model.

In order to make use of the AA's services, application developers must build the application based on the programming interface provided by AA. The application's binary file has to be compiled with the AA library. Developers can perform three basic operations through the APIs provided: request nodes, release nodes, and replace nodes. These features allow an application to request additional nodes, and to remove existing nodes, or to replace nodes at any time during the execution without knowing how they are discovered and allocated. Since processing demands may change drastically during the lifetime of the job depending on the results at the end of its completion, an important aspect of the AA's functionality is that the resource requirements (e.g., job execution time, requested number of resource nodes etc.) of an executing job do not have to be anticipated in advance by developers.

The AA comprises several core services: `LComm`, `PMWComm`, and `RD`. The `LComm`, `PMWComm`, and `RD` take charge of local communication, process management/wide-area communication, and resource acquisition, respectively (Liu, Nazir, and Sørensen, 2009). For ease of use, the programming interface presents the familiar interface of PVM (Beguelin et al., 1991), which is a common message passing interface that has been used by many scientific applications. For example, the AA provides the application programming interface (API) which allows the application developers to start a named process during the execution by `RequestProcessors(Requirements)`, which is a non-blocking function.

Each function call should be associated with *specification.xml* file, which specifies the resource requirements to run this process. The requirements include the estimated job execution time, job memory requirement, offered monetary value, and a deadline constraint. The AA translates the application resource request into an XML file consisting of the quality-of-service (QoS) parameters. An example XML file is shown in Figure 4.3. From Figure 4.3, we can see that resource request in XML format contains many of the job requirement parameters including the job estimated runtime, job memory requirement, job offered monetary value, and job deadline. HASEX parses these parameters and uses them to perform the appropriate scheduling and renting decisions. If resource node information (i.e., node architecture and node speed factor) is not explicitly specified, HASEX will assume that a job can be started on any (standard) resource nodes.

As soon as the `AddRequest()` function is invoked, HASEX will send a rental request and start negotiating for a resource node from one or multiple distributed resource managers (DRMs). The `AddRequest()` is a non-blocking request which does not interrupt any running application during the resource negotiation process. The time it takes to be allocated with a node is not bound. Therefore, application developers must embed the `GetNotification()` with `NODE_ADDED` tag in order to receive notification when the new node addition has been allocated to the application. If a node is successfully allocated to the application, the `GetNotification()` will return true. The Unique Node ID (UNID) is generated and is used to identify the allocated node for communication purposes.

Finally, the `TerminateNode( UNID )` call can be invoked to release a node before its allocation period ends. Upon this invocation, the rental contract of a given `UNID` is terminated and subsequently the resource node is returned to the resource owner (external provider).

A full description of the API can be found in (Liu, Nazir, and Sørensen, 2009). An example of API usage is also shown in (Liu, Nazir, and Sorenson, 2007).

```
<? xml v e r s i on =" 1 . 0 "?>
<Request>
<Requirement name="Runtime " type ="<=" value =" 600 " />
<Requirement name="Memory " type ="<=" value =" 8000 " />
<Requirement name="Value " type ="<=" value =" 10 " />
<Requirement name="Deadline " type ="<=" value =" 1200 " />
</ Request>
```

Figure 4.3: Resource requirement of job embedded in XML file.

4.2.3 VA Scheduler

The core functionality of VA Scheduler is to keep track of all incoming jobs in the queue and to schedule these jobs on 'rented' nodes. How the nodes are rented will be discussed later in the next section. For now, the VA Scheduler is responsible for job scheduling and resource allocation policies. It keeps track of and maintains jobs' statuses and retrieves their output if required for use by standard batch and workflow systems. The VA Scheduler is comprised of six services which are described in the following subsections.

4.2.3.1 Allocator

The HASEX's Allocator or HAllocator identifies suitable physical servers for job scheduling. It makes resource allocation decisions by determining the nodes the jobs should be assigned to. Each time a node is allocated to a job, a resource node permit is generated for the application. A permit is a proxy credential that gives permission for the application to access, submit, execute, and control its jobs on its allocated nodes. It provides information on the expiration period, i.e, the access period – the maximum duration the application is given access to its allocated node. After the access period expires, the permit is no longer valid and the application no longer has access to its allocated node. The HAllocator clears the data and purges files from the execution node as soon as the permit expires.

When a permit is granted, HASEX formulates computation-specific allocation functions as a set of application 'tasks'. A generic `RequestTask` operation is invoked to generate an application request, according to the supplied arguments. Each `RequestTask` call returns a handle that can subsequently be used to monitor and control the task. Furthermore, the application agent also receives notifications on specific actions or events, such as task creation, task termination, or QoS contract violations. Handling each request as a task enables the system to provide rapid response to sudden change in application processing requirements so as to maintain specific performance requirements, such as smooth time progression of execution (Liu, Nazir, and Sorenson, 2007).

The HAllocator also provides the option for the application developers to be notified when allocated nodes are about to expire. This is carried out via the invocation of the AA's `GetNotification(UNID)`. Although the `GetNotification()` function sends a notification message to the AA before the period expires, it is the responsibility of the AA to vacate the execution result and the output data on behalf of the application before the results are being cleared without any further notice. In such way, the HAllocator can schedule, allocate and reuse the same node for multiple applications and jobs without compromising security issues.

4.2.3.2 Monitor

HASEX's Monitor or HMonitor monitors the health and status of rented nodes once they are deployed. It keeps monitoring the status of rented nodes for charging and billing purposes. HMonitor tracks resource node faults and resource node consumption across all rented nodes that are inherently failure prone. Unlike the AA , the HMonitor does not keep track of individual job performance, but monitors the total execution of jobs to compute the level of utilization of each node.

4.2.3.3 Pricer

HASEX's Pricer or HPricer records and keeps track of the monetary values received from applications for executing their jobs. When a job is admitted to the system, the HPricer keeps track of this job and waits until the job completes before it computes the total monetary value earned from the job.

4.2.3.4 Participator

Compute nodes in HASEX are dedicated blocks of CPU time used to run application jobs to completion. The participator or HParticipator provides a secure execution environment for the applications using an execution sandbox[2] model. A resource node is fully dedicated to one application at any one time. When the HAllocator grants a resource permit to an application, the HParticipator provides an exclusive environment for the application to be guaranteed access to its allocated node without any interruption from other running applications in the system. This is important to ensure performance isolation. Hence, in practice, both the resource permit and the execution sandbox enable HASEX to provide a hard performance guarantee for job execution since arbitrary sharing cannot occur.

The HParticipator is normally deployed when a newly rented node is first configured. When a rented node is first configured as part of HASEX, a rental duration is specified by the HParticipator. The HParticipator resides on a rented node until its rental duration has expired.

In rare situations, the communication between rented nodes may fail, even if the HParticipator has been successfully configured. This may occur if the rented node becomes unavailable due to a sudden change of resource states (e.g., network problem, node shutdown). However, this problem can be resolved by enabling the HParticipator to send 'ALIVE' messages continuously at some periodic interval in order to notify the system that a rented node is still available and active.

4.2.3.5 Controller

HASEX's Controller or HController manages and synchronizes network and node information such as Unique Node IDs (UNIDs), hostnames, and IP addresses of all rented nodes. The HController also maintains each node's network configuration files (e.g., Linux host file, PVM or MPI configuration files).

The HController prepares a set of master configuration files for communication purposes to link all rented nodes for the deployment of parallel applications. The same master configuration files are initially copied, distributed and configured to each rented node. This enables resource nodes to recognize

[2]The execution sandbox model allows the description of a code execution model as well as a set of rules used to develop code that will execute in a well-defined environment. A sandbox can be used to execute untrusted code on a machine, while having access to a set of system functionality such as network connectivity, while protecting the executing machine as well as the data attached to it from any security attack that could occur from malicious or deficient code loaded by the sandbox.

```
RequestRent -lid < string > [-cores <n>] [-os < string>] [-memory < n >] -disks < string >
CancelRent -lid <n>
RequestTask -tid < string > [-permit < string > ]
CancelTask -tid <n>
RegisterEvent -eid [-event <n>]
ModifyRent -eid [precond < string > ] [postcond < string > ]
```

Figure 4.4: Listing of sample HASEX Rental API functions.

each other in order to support the execution of parallel applications. For future retrieval, the master configuration files are stored in the resource repository.

Each time a new rented node arrives, the HController retrieves the master configuration files from the resource repository and updates the master configuration files appropriately with the new node information. Then, the master configuration files are sent across the network to each rented node and subsequently each of the relevant network files of a resource node are updated with the new updated master configuration files.

4.2.3.6 Admission

HASEX's Admission or HAdmission validates and verifies whether the job comes from registered and authorized users. It also checks whether the job contains sufficient information (e.g., number of requested nodes, deadline information, monetary value etc.) for the HAllocator service to perform scheduling.

4.2.4 VA Negotiator

In this section, we describe the core services of the VA Negotiator in more detail. The VA Negotiator is a core component that takes charge of renting decisions, resource discovery, establishes wide-area communication between rented nodes, and manages groups of rented resource nodes across multiple clusters and sites.

4.2.4.1 Rental

The HASEX rental interface is an extensible interface that includes API calls to create and cancel rent requests, and to query and to request notification of changes in the status of rental requests. A sample listing of the HASEX Rental API is shown in Figure 4.4.

A `RequestRent` call takes as input a representation of the required nodes and returns a rental request handle, while a `RequestTask` call takes as input a specification of the required task and a rental handle representing the nodes with which the task is to be associated, and returns a task handle. Both `CancelRent` and `CancelTask` allow cancellation of previously requested rentals or tasks, using the appropriate handles. `RegisterEvent` allows an application to request notifications on selected events, such as the failure of a rental request. `ModifyRent` allows an application to adapt if existing rental requests cannot be honoured or when an application's requirements change. Besides the API function calls provided by HASEX, the system can use rent request and task specifications to control various aspects of the rental behaviour (e.g., about what should happen to a node when its rental duration expires).

The HASEX rent interface provides the high-level functionality required for applications to submit, control, and monitor jobs in local cluster DRM systems. HASEX services are deployed at the remote resource DRMs, instead of using the interfaces commonly used in Grid systems (e.g., WMS, GRAM etc.)[3]. It takes a rent request as input and authenticates the rent request. The authentication mechanism is provided by the standard UNIX environment including SSH and RSH. If the authentication is successful, HASEX interfaces with local DRMs to submit a 'rent' request, and returns a 'rent handle' as output. The rent handle is used to monitor and control the state of a task computation and can request notifications for specific events, e.g., rental cancellation, termination, etc.

The Rental service or HRental initiates the renting process when it receives the `RequestRent()` call from the HPlanner. When receiving such a request, its role is to handle the rental request and manage the rental of a rented node until the rental has expired. Each time a node is rented, the HRental stores the information of the new rented node into the rental repository, i.e., its assigned UNID, hostname, IP address and its rental duration. It then sends a request to HComm to configure the rented node. Moreover, the HRental also sends a request for the HParticipator service to be deployed on a newly rented node. HParticipator will continue to reside on the node until the node rental has expired.

A rental request is created by a generic `RequestRent` operation, which interacts with DRM entities to ensure that the requested quantity and quality of the node will be available and will remain available for the desired rental period. All `RequestRent` operations require a reservation in order to proceed. This reservation is normally created with a guaranteed reservation policy, although the '"best effort"' reservation can also be supported. In HASEX, we are primarily interested in immediate reservations, which are assumed to be follow oned immediately after a rental call or invocation, unlike GARA (Foster et al., 1999a) which reserves resources for use in the future. Nonetheless, HASEX can also be extended to support future reservation.

Most DRMs (e.g., SGE, LSF, MAUI PBS, etc.) guarantee that all accepted reservations are to be committed. This implies that a rental request cannot be cancelled before its expiration period. On the other hand, HRental may release a node earlier than the reservation time it was previously granted. Furthermore, HRental will have the option to request for rental renewal, if directed by the HPlanner service, before the expiration period. However, there is no guarantee that the renewal request will be granted by the external resource provider for some reasons. For example, the resource provider may reject the new reservation request due to resource unavailability.

When the rental duration has expired, the `ReturnRental(unid)` call is returned. The function call results in the HParticipant with the given UNID being terminated and the expired node is returned to the external resource provider immediately.

4.2.4.2 Discovery

HASEX's Discovery service or HDiscoverer performs resource discovery and allocation activities by contacting the external resource providers worldwide. HDiscoverer negotiates nodes with external resource DRMs (resource providers) via the HASEX Rental API. The HDiscoverer initiates prior agree-

[3]In Appendix C, we present IntelligentGrid, another prototype based on HASEX architecture that enables one to rent resources in a Grid environment without the need to modify existing configuration of a Grid production system.

ments with the external resource providers to rent their nodes in times of high demand.

The HASEX Rental API translates each rental request into a specific advance reservation query that conforms to either MAUI PBS, SGE, or LSF. In doing so, the HDiscoverer configures set of accountable credentials across multiple DRMs (resource providers) to support personal reservations on each DRM. A personal reservation is needed because HASEX requires a permission account to access resources from an external resource provider before it can start accessing the respective resource nodes of a resource provider.

HASEX decouples fundamental rental mechanisms from scheduling and resource allocation policies. For the rental phase, a reservation is requested from an external resource node. Once a node is rented, no task is created at this time, Instead, a rental handle is returned which is issued to the AA. The rental handle is used by the AA to monitor and control the status of a rented node.

When a rental request is initiated, the HDiscoverer contacts multiple external resource providers in parallel and sends them reservation requests. The `RequestRent` call hides the complexities of negotiating with different resource providers that have different underlying resource management architectures (e.g., SGE, LSF, PBS etc). Simply, the HDiscoverer provides a mechanism that translates a rental request into a set of reservation resource requests that conform to the specific reservation protocols of the resource provider domains.

A resource reservation request typically contains information of its start time (start rental time) and its end time (expiration time). It may also contain the resource description, and the QoS requirements. From these reservation requests, the HDiscoverer simultaneously negotiates nodes with multiple resource provider domains. However, for one rental request, only one reservation request is submitted to each resource provider domain in order to avoid the potential traffic bottlenecks.

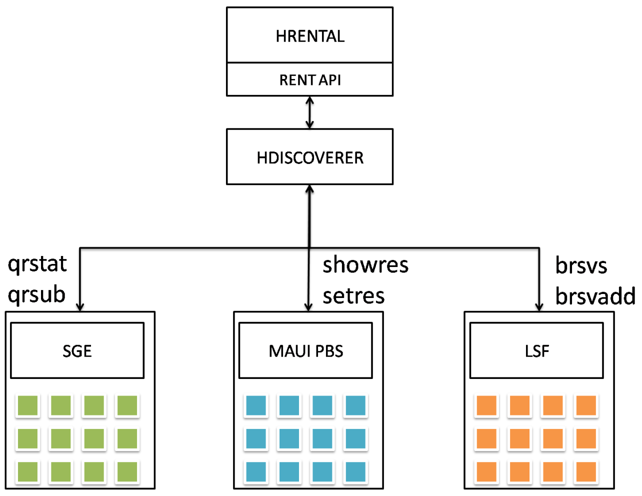


Figure 4.5: The discovery process of renting a node.

Current DRMs such as PBS, Torque, and SGE perform resource allocation along with job deployment, but they do not normally give total access permission to the third-party software (Liu, Nazir, and Sørensen, 2009). Therefore, the HDiscoverer also provides the communication links for establishing, maintaining and securing communication between rented nodes belonging to different resource providers. Figure 4.5 shows how the HASEX's HDiscoverer communicates and obtains the nodes from external resource providers, which are controlled and managed by individual DRMs such as SGE, Maui PBS, and LSF. As shown in the figure, the HDiscoverer initiates specific commands based on the DRM type when requesting resources.

4.2.4.3 Planner

HASEX's Planner service or HPlanner obtains, periodically, the historical resource usage information, or job execution historical information from the monitoring service (i.e., HMonitor), and uses this information to construct predictions of future resource availability and future expected load. The HPlanner is one of the most important services for the VA Negotiator because it is responsible for triggering rental decisions: when exactly to initiate a rental request for additional nodes, and when to release rented nodes. For example, it may trigger a rental request when there is a job waiting in the queue (i.e., queued job). Similarly, it may trigger a release request when the number of idle rented nodes have reached a specific resource level threshold.

4.2.4.4 Communicator

Nowadays, many computational clusters are configured with only one IP visible frontend that hides all its internal nodes from the external world (Petrone and Zarrelli, 2006). In these network architectures, it is difficult to exploit the nodes, since they are located in a different network domain that is geographically and administrative separated, usually shielded by firewalls. In addition, some of the clusters are configured to use network address translation (NAT) and have private networks that are not accessible to external nodes.

To resolve the issue of wide-area communication, the Communicator service or the HComm is deployed on each rented resource node in order to establish a communication link between the main HController and all rented nodes. The HDiscoverer must first initiate prior agreements with the external providers to rent their nodes in times of high demand. When agreements have been made, the HComm is deployed to each frontend node of the resource provider. Subsequently, the information of the frontend node is stored into the rental repository for future retrieval by the HDiscoverer. Figure 4.6 shows how HDiscoverer, HController, HComm, and HParticipator play their roles in establishing and maintaining wide-area communication links between rented nodes and the resource provider. As observed, the HComm needs to be deployed on the frontend node of a resource provider as the main proxy to route messages and manage resource nodes over a wide-area network. Once the HComm has been deployed, the HController routes the messages to the newly rented nodes and to all other rented nodes in order to connect them as a single scheduling entity.

4.2.4.5 Accounting

After a new node joins the system, the HASEX's accounting service or HAccount is activated and starts to monitor and records the rental activities (node resource information). Such information includes a fraction of jobs successfully executed on the node and the utilization of the node over different time scales. The information is stored into the rental repository for future retrieval and HAccount continues

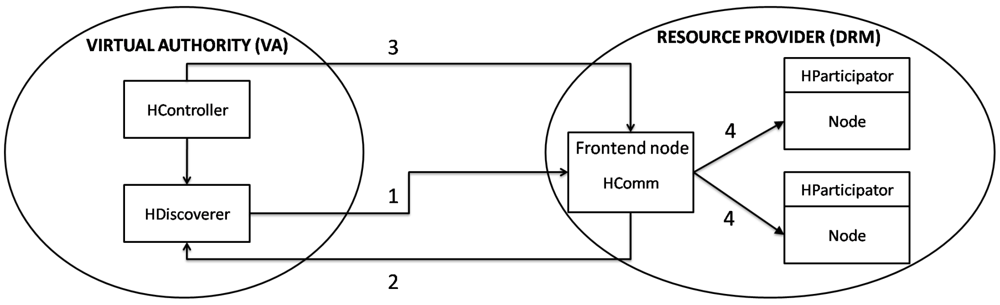


Figure 4.6: The transparent remote execution is achieved by HDiscoverer, HController, HComm, and HParticipator.

to monitor the node activities until its rental duration has expired.

4.3 Virtualization

To address the heterogeneity issues, HASEX is built on virtualization technology. Virtualization technology provides distinct comparative advantages over the purely physical environment due to the increased integrity, isolation and security it offers. It offers HASEX the solution for enabling resources to be customized and delivered on demand.

The virtualization mechanism enables the resource provider to customize its hardware with the necessary resource requirements requested by the service provider (e.g., memory capacity, storage capacity, software execution environments). For example, a physical server without virtualization technology can only deploy one operating system (OS) per physical node. It is not possible to replace different OS on a physical node without manual administrative intervention. On the other hand, the virtualization feature removes this restriction by allowing one to deploy multiple operating systems on a physical server in the form of virtual machines (VMs). The VM can be customized with a specific operating system and the applications and/or services required by the service provider.

Furthermore, in the purely physical environment, the scheduling application may fail due to non-availability of software execution environment (e.g., non-availability of specific applications, libraries, programs etc). With virtualization, the VM can be customized with the relevant software execution environment for job execution. Therefore, the problem of the non-availability of the required execution environment for an application can be resolved with virtualization.

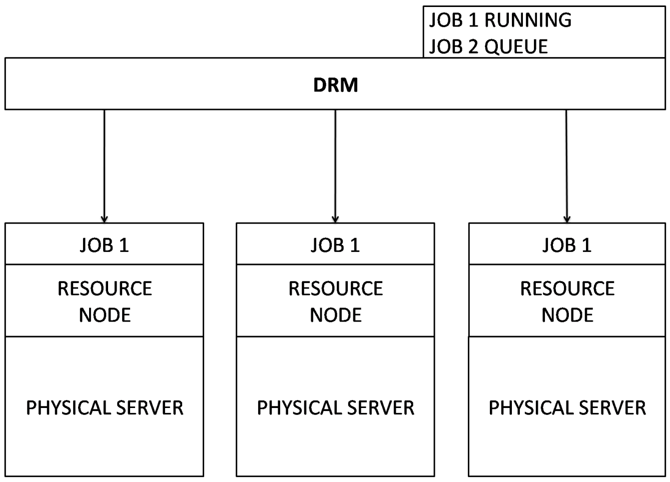
Most importantly, virtualization makes it possible to get the isolation required to maintain the separation between multiple jobs because virtual machines are isolated from each other as if physically separated. This enables each VM to be tailored specifically for each job execution environment. In summary, virtualization makes it possible for HASEX to provide support for resource renting.

4.3.1 Resource Renting

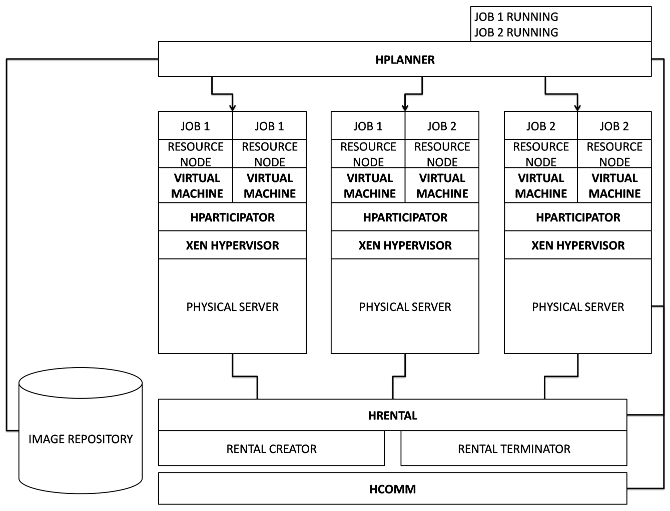
In this section, we present the resource renting mechanism of HASEX. Effectively, there is a need for a mechanism to deploy resource nodes on demand within a suitable environment for job execution. HASEX offers this ability by using virtualization technology based on the XEN (Barham et al., 2003a) hypervisor.

Virtualization is handled by the HRental service that assigns virtual machine images to physical server machines. The HRental service controls booting by opening a secure connection to the privileged control domain using Libvirt (Bolte et al., 2010) on the XEN node, and issuing commands to initiate and control XEN virtual machines. Libvirt provides a hypervisor-agnostic API to securely manage guest operating systems running on a node.

The combination of support for both physical and virtual machines offers useful flexibility: it is possible to rent out resource nodes via the conventional access to clusters and/or through direct access to physical server machines by using virtual machines. The system can be deployed easily without needing to reconfigure the DRM. The only requirement is that the XEN be installed on physical servers with the appropriate virtual machine deployment permission. The rest of the configuration occurs dynamically, through the interactions of various HASEX components. By simplifying configuration, HASEX makes it possible to automatically and dynamically provide additional compute nodes without user intervention.



(a) Conventional Resource Management



(b) HASEX with virtualization.

Figure 4.7: HASEX uses XEN hypervisor to support virtualization.

The HRental further consists of two core sub-services, namely, Rental Creator and Rental Terminator. Figure 4.7a shows a conventional deployment scenario under a pure physical environment with three physical server machines whereby two jobs are initially submitted to the HASEX pool with each job requesting three nodes. Assuming that jobs are served in a FCFS manner, we can see that Job 1 is executed immediately while Job 2 is queued due to insufficient nodes.

Figure 4.7b shows how the renting mechanism is provided by HASEX so that additional resource nodes can be deployed. Similarly, it can be seen that the HASEX pool consists of three physical server nodes with a HParticipator deployed on each resource node. However, with the same amount of physical server machines, HASEX makes it possible to deploy three additional nodes on each of the physical servers for the execution of Job 2 via virtualization support.

Figure 4.7b also shows a number of additional HASEX services that are introduced to support on-demand provisioning of resource nodes. The HRental keeps track of all jobs, all virtual machines that have been created and terminated by HASEX, and all virtual node specification information (i.e., number of nodes, CPUs, memory, and storage for each VM). Initially, the creation of a new VM is triggered when a resource request from the HPlanner is received. The Rental Creator acts when it receives an event from the HPlanner. Subsequently, a number of VMs are created based on the number of requested nodes, CPUs, amount of memory and storage capacity requested by the job.

The Rental Creator has a number of functionalities. First, the Rental Creator identifies a subset of the most suitable physical server machines to deploy the additional VMs (i.e., three VMs in our case). The physical server is selected in a round-robin fashion to balance the resource load on each physical server. Second, it extracts the job requirements to identify the relevant VM image that fulfils the job requirements for execution. The VM image is either retrieved from the image repository or it may be accessible directly from the AA. Third, a new hostname and IP address are obtained from the HDiscoverer for the configuration for each new VM. Upon successful creation of each VM, the newly configured resource node information is updated.

To support parallel applications such MPI and PVM, the HComm prepares the network configuration and the parallel communication configuration file necessary for the execution of parallel jobs. These configurations files are copied across to each rented node (i.e., virtual machines and physical nodes) through secure channel communications so that parallel communication between remote nodes can be established. Identical network and parallel configuration files are used across all rented nodes so they can recognize and communicate with each other. After the network and parallel communications files have been configured, nodes are allocated to the job for the execution. Finally, when the rental period has expired, the HRental notifies the Rental Terminator to remove the entry of the VM information from the master network and parallel configuration files. Subsequently, the HComm synchronizes and updates the network and configuration files to all other running virtual machines across the network via secure channel communications.

4.4 Implementation and Deployment

In this section, we describe how we implemented HASEX based on the core components introduced in Section 4.1.

All HASEX services are currently implemented in Java for portability except for a few communication functions written in C for compatibility. The AA library is implemented in C++ for speed. Communication security is based on SSH. File transfers are performed by SCP.

We used the Apache XML-RPC implementation for communications between various HASEX services. XML-RPC is a Remote Procedure Calling protocol that uses XML to encode its calls and HTTP as a transport mechanism. We chose XML-RPC instead of other protocols (e.g., CORBA, SOAP, RMI) because of its ease of use, and it is a platform-independent remote procedure call (RPC) protocol. Also, the HTTP-based protocol is more firewall-friendly, which enables HASEX to deploy proxies to resources over the Internet without specific configurations. The common communication interface is also implemented to provide flexible access to established communication methods such as PVM and MPI. This makes it possible to connect a large number of geographically distributed and heterogeneous resources.

For the HAllocator, we used the Sun Grid Engine (SGE) job scheduler and resource management system. All incoming job requests are handled and managed by the SGE. However, the default SGE scheduler can also be replaced with any other Distributed Resource Manager. Integrating HASEX with SGE is straightforward and required no modifications to SGE itself to support a rich set of resource management capabilities.

The HAdmission also resides on the same physical machine as the HAllocator. When a new job arrives, the Job XML specification is parsed by HAdmission. Based on the offered job value and minimum threshold, it determines whether to accept or reject the job. The AA is notified if the job is being rejected. If the job is rejected, it will not be scheduled by the HAllocator.

In order to provide transparent and remote execution of nodes, the HParticipator must support interfaces to different DRMs (e.g., LSF, SGE and PBS). At the moment, only interfaces for SGE and Maui PBS are implemented and supported. However, HASEX can be extended further by developers to support alternative DRMs for future extensions. Furthermore, disk images for SGE and PBS were also configured and prepared in advance for the deployment of virtual machines. The virtual machines will be deployed as resource nodes to support resource renting.

The HController is an XML-server that is triggered each time a new node is rented or when a node has expired. The HController updates the host files, and the message passing libraries (i.e., PVM and MPI) configuration files, which are necessary to support the exclusive environment for the parallel execution environment between rented nodes. These files are updated as master configuration files. The updated master configuration files are then sent across each rented node that belongs to the system.

An HComm service communicates to external services via XML-RPC communication, while each HComm communicates to its internal HParticipators via alternative communications, which can be from PVM or MPI. The remote invocation of HComm is performed through the SSH protocol.

The external resource provider is built on SGE's DRM via a common interface. This interface is integrated with the HASEX Rental API. The rental aspects are managed by SGE. Furthermore, user reservation and space-sharing features are enabled on the SGE for personal reservations. This allows each application to create, modify, cancel, and query the reservations it owns. As with jobs, applications may associate a personal reservation with any QoS or account to which they have access. This provides hard performance guarantees to job execution because arbitrary sharing is not permitted between different applications.

The HDiscoverer service communicates with HParticipator services via the XML-RPC protocol. Initially, the HDiscoverer contacts the DRMs through the HComm services. A HComm is then placed on the frontend (head node) of each cluster DRM. The HComm implements the advance reservation capability by reserving time slots as queues on the external provider. Before the HComm can be deployed on the frontend node of the resource provider, it needs to have a personal user account created on the SGE's frontend node machine. The created user account will give all the necessary permissions for the application to deploy proxies on individual resource nodes. This is necessary to provide a parallel execution environment for interactive steering applications (RPC-, PVM-, and/or MPI-based applications).

When a resource reservation is made by the HDiscoverer, the HComm creates a personal reservation account by using the command line interface provided by the SGE and returns the name of the created reservation queue as the reservation application ID. Once a node has been allocated, the newly-created reservation queue is tied to that specific allocated node. When the reservation period starts, the reservation queue is activated for future job arrivals. At this stage, this reservation queue is attached to the HComm service until the agreed rental period expires. When the rental period has expired, the SGE scheduler terminates, deactivates and removes the personal reservation account and the reservation queue.

4.5 Evaluation

We have constructed implementations of the HASEX core services described above, and have demonstrated the ability of HASEX to perform rental requests across multiple resource providers with different resource types. Here, we present experimental results that provide insights into the performance of our HASEX system and its implementations. We compare the performance of the HASEX implementation with that of the gLite Grid middleware (Codispoti et al., 2010) to examine how HASEX performs relative to an existing Grid production system (a global scheduling approach). The gLite middleware is chosen for comparison because it is currently the most widely used and reliable Grid middleware system. The gLite middleware is used for the development of EGEE Grid production which is currently the world's largest production Grid.

We are interested in answering three questions: (1) How scalable is HASEX in providing support for resource renting? In particular, what are the costs of our rental requests, i.e., deployment and configuration latency and resource (node) allocation mechanisms? (2) How effective is HASEX in providing a mechanism for cost-aware rentals? In particular, how do these costs compare with the current global resource sharing approach (Grid approach) if nodes are allocated and scheduled by the Grid scheduler?

(3) What does a comparison of the rental request and task allocation costs tell us about the practical aspect of the rental-based strategies employed by HASEX?

First, we assume a typical Grid environment where a meta-scheduler (Grid scheduler) is used to coordinate heterogeneous and geographically distributed resources from multiple DRMs worldwide and where these resources are integrated as one global scheduling entity. The main objective of such a Grid Scheduler is to perform load balancing across multiple DRMs for the purpose of increasing performance, resource utilization, or response time. The Grid Scheduler is also in charge of managing reservations for workflow jobs.

As a main component of any Grid system, the Grid scheduler interacts with other components of the Grid system: Grid information system, cluster DRMs, and network management systems. It should be noted that in Grid environments, multiple DRMs coexist, and they could, in general, pursue conflicting goals. Therefore, there is a need for information synchronization between the different DRMs in order to perform scheduling and resource allocation efficiently.

The Grid scheduler follows a series of steps which can be classified into five actions (Xhafa and Abraham, 2010): (1) preparation and information gathering on tasks, jobs or applications submitted; (2) resource selection; (3) computation of the planning of tasks (jobs or applications) to selected resources; (4) task (job or application) allocation according to planning (the mapping of tasks, jobs or applications to selected resources); and (5) monitoring of task, job or application until its completion.

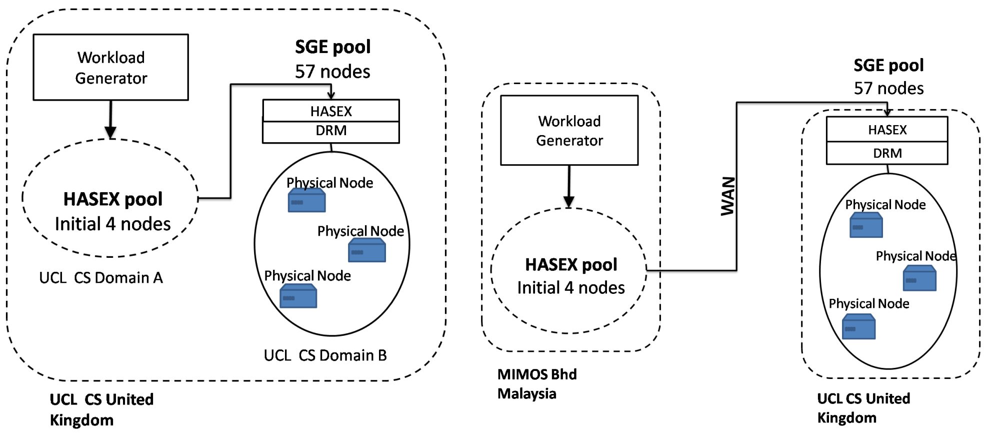
For the Grid middleware architecture, gLite (Codispoti et al., 2010) is chosen because it has been used to develop the world's largest Grid infrastructure which is currently adopted by more than 250 Computing Centres worldwide (Berlich et al., 2006). Furthermore, gLite provides a complete set of services that simplifies our development for a Grid test-bed.

4.5.1 Test-bed Environment

Figure 4.8 shows an overview of the experimental test-bed. The test-bed comprises two geographically disparate sites which are controlled and managed separately by different administrative domains. The main site is located in London, UK and is managed by the Computer Science Department of University College London (UCL CS). Another site is located in Kuala Lumpur, Malaysia and is managed by our Southeast Asian partner, MIMOS Bhd.

For the purposes of our experiment, we have two different test-bed implementations: HASEX LAN (Figure 4.8a) and HASEX WAN test-beds (Figure 4.8b). The HASEX LAN test-bed is set up to test the performance of HASEX services between nodes belonging within the same local area network (LAN). The HASEX WAN test-bed is set up to measure the overhead of wide-area network communication in the test-bed, i.e., interaction of HASEX services from one site with resource nodes at another geographically disparate site consisting of hundreds of resource nodes.

The HASEX experimental system is located at MIMOS Bhd and it is initially set to 4 resource nodes only, each equipped with a Dual-Core AMD Opteron(tm) processor. The UCL CS site, however,



(a) HASEX LAN Test-bed. (b) HASEX WAN test-bed.

Figure 4.8: Test-bed architecture.

has a much bigger resource capacity. It manages 57 nodes, each equipped with Dual Quad Core Intel(R) Xeon(R) 2.66GHz or Intel(R) Xeon(R) CPU 2.80 GHz processors, RAM from 2 GB to 30 GB, running Red Hat Enterprise Linux. The 57 nodes are managed by the SGE distributed resource management system (DRM). The cluster is a sub-domain of the department. The SGE provides a frontend node that interfaces the other resource nodes to the outside, and it is configured by network address translation (NAT). Therefore, in our experiments, the SGE DRM at the UCL CS represents a potential resource provider with which the HASEX system communicates and negotiates for resources.

With regards to the Grid test-bed, the Grid architecture contains a Resource Broker (RB), a virtual organization management server (VOMS), a resource information harvesting and integration engine, a number of resource information index servers for Linux- and Unix-based resources, and finally a single sign-on authentication service through gLite's MyProxy (Basney, Humphrey, and Welch, 2005) service. The WMS (Workload Management System) provides sophisticated job-resource matchmaking mechanisms, allowing the dynamic requirements-based job allocations to be performed (Figure 4.9). For the experiment, the WMS points both to a small SGE cluster at MIMOS Bhd (4-node capacity) and also to the SGE cluster at UCL CS (57-node capacity).

4.5.2 Basic Performance Test

The first experiment is a simple one, where we are primarily interested in measuring (1) the amount of time it takes for a job to be served and (2) the amount of time it takes for a job to be allocated to a rented node as soon as the rental request is initiated. Specifically, for this experiment, we are not interested in measuring the network latency across geographically disparate sites. Therefore, our experimental set-up only involves one site in London, UK, but the site is administered by two different domains: UCL CS Domain A to UCL CS Domain B (Figure 4.8a).

The test assumes that the difference in latency caused by the network itself (bandwidth, distance, and traffic) can be ignored. Using the workload generator, we submit a single job request that requests

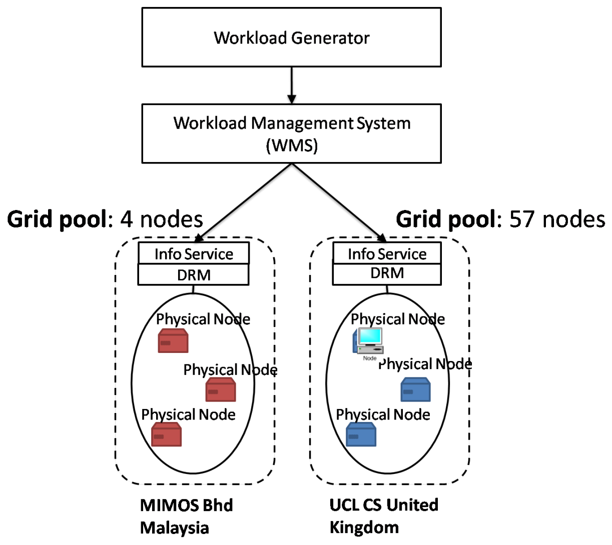


Figure 4.9: Grid test-bed.

one node only. Since we are not interested in measuring the job running time, each job submitted will only sleep for approximately half a minute (30 seconds) and finish; it does not produce or require any input and/or output data files.

Timings are measured from two points: (i) the time it takes for the HAllocator to identify a queued job and initiate a rental decision as soon as the job is queued due to insufficient resources and (ii) the time it takes for HDiscoverer, HComm, and HController services to configure a rented node so that it is ready for allocation. We timed certain major HASEX operations, measuring the cost of individual operations, as well as the time required to make rental requests. The HAllocator represents the time it takes to allocate requests upon new node availability; the HDiscoverer represents the time it takes to discover physical nodes from the resource provider (i.e., SGE DRM); the HComm represents the time it takes to configure a virtual network between two different networks (i.e., rented nodes from UCL CS to MIMOS Bhd); and the HController represents the time it takes to prepare the DRM and parallel configurations in order for newly rented nodes to be remotely accessible to a local site at MIMOS Bhd.

We first measure the mean wait time for intra-domain communication (interaction between two nodes of different DRMs but within the same gateway). The average wait time includes the time it takes to trigger the rental requests for queued jobs, the time it takes to request external nodes, and also the time it takes to configure and deploy new nodes. The wait time metric is chosen instead of the average response time because the former ignores the impact of heterogeneous resources when performing scheduling.

The results are presented in Table 4.1. As observed, the overhead of renting can be considered minimal, and is primarily dependent on the time it takes for the DRM to process a reservation request. Once the reservation request confirmation has been returned, the time it takes for HDiscoverer and HComm to configure the reserved node is negligible, as can be witnessed by examining the configuration delay. The configuration delay represents the amount of time elapsed between the process of HDiscoverer at UCL CS Domain A and the process of HComm at UCL CS Domain B. This is primarily due to processing

	Basic Test (LAN)	Stress Test (LAN)	Stress Test (WAN)
Total Number of Requests	1	1000	1000
Inter-arrival (secs)	60	exp(60)	LLNL Thunder Trace
Job Length (minutes)	0.5	normal(30,7.5)	LLNL Thunder Trace
Allocation Overhead (secs)			
HAllocator	5.2	6.1	8.4
Rental Overhead (secs)			
HDiscoverer	4.2	4.7	71.4
HComm	29.6	124.9	143.5
HController	34.6	36.8	53.8
Total Overhead (sec)	73.6	172.5	277.1

Table 4.1: HASEX performance test.

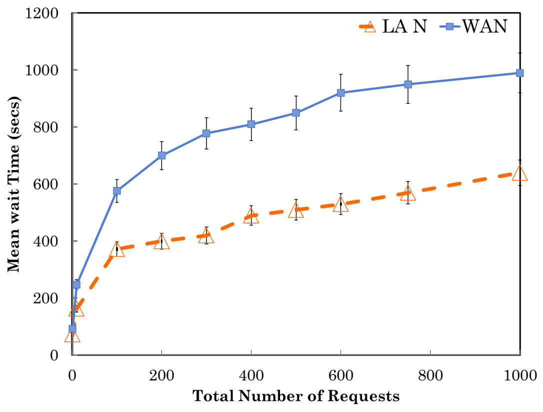


Figure 4.10: Stress performance test (*n=20*).

delays of the DRM rather than the overheads incurred by the HASEX services.

4.5.3 Scalability

To demonstrate the incremental scalability of the system, we aim to investigate the performance of HASEX under a sudden burst of job requests at peak times. To simulate this effect, instead of submitting a single rental job, we submit *n* number of jobs at exponentially distributed intervals with a mean inter-arrival time of one minute (60 seconds). In this way, we can stress the system under multiple job requests and examine the impact of an increased number of job requests on the overall mean wait time. For the experiment, HASEX is initially configured with 4 nodes from MIMOS Bhd, but it dynamically rents additional compute nodes from the SGE DRM which comprises a resource pool of 57 nodes.

The results are shown in Figure 4.10 for repeated experiments under varying total numbers of submitted requests. As can be observed, the mean wait times increase as the number of submitted job requests increases. Interestingly, for 1000 simultaneous job requests, HASEX can serve all 1000 job requests in under less than 1000 seconds. Our initial observation is that the mean wait time increase is fairly high when the total number of requests is between 1 and 100. However, as the total number of job requests rises above 100, increasing the total number of requests has only a very little effect on the mean wait time. We can see that there are no significant performance differences under increasing total number of job requests.

Figure 4.10 also shows the performance difference between local area network communication and wide-area network communication. We can observe that the difference in mean wait time between LAN and WAN is fairly high within 52% range. The delay is attributed to the overhead of the HDiscoverer in establishing communication with the cluster DRM over long distances from source to destination. However, once communication has been established and a personal account has been created, the delay is minimal for subsequent requests. This shows that the bottleneck of the system is primarily due to the authentication and the interaction with the DRM (i.e., to request a personal account on external resource nodes) rather than due to the renting process itself (i.e., the process of establishing communication links between HASEX services on rented nodes). This is attributed to the complex protocol of message exchanges between the HDiscoverer and the DRM.

The conclusion that can be made from the result is that the process for the deployment of a new virtual machine on a Cluster DRM is more costly than the cost to establish wide-area communication links between nodes. However, the overhead is still negligible since an additional latency of 5-10 minutes has very little impact for most long-running jobs that take several hours to complete. For jobs that require many nodes at short bursts and run only for short durations, the overhead can be a problem. However, this can be resolved by employing a more sophisticated rental policy that predicts the amount of resource nodes to be rented in order to accommodate deadlines of current and future requests satisfactorily. Nevertheless, we find these results encouraging in terms of what they reveal about HASEX's scalability.

The HASEX design supports the notion of a pool of overflow nodes that are not dedicated to the system, but are recruited by the system only in the event of extreme bursts. These nodes can be machines that are usually used for other purposes (such as workstations), or machines that the service can 'rent' on a pay-per-use basis from a resource provider. When the system runs out of dedicated machines to handle the system load, it may rein in idle machines from the overflow pool and launch additional nodes on those machines. As soon as the load on the system subsides, the overflow machines are released.

4.5.4 Cost-Effective Resource Management

One of the key features of HASEX is that it provides cost-effective resource management. Cost-effective resource management implies the ability of HASEX to complete job requests with high QoS satisfaction while lowering the infrastructure and operating cost. For evaluation, we compare the performance of HASEX to that of a conventional Grid system. The Grid system serves as the baseline for comparison since it is currently the standard mechanism employed to connect multiple, worldwide resource providers that are owned by different authorities.

For a more realistic evaluation, we perform the experiments using real High Performance Computing (HPC) workload traces obtained from Feitelson's Parallel Workload Archive (D.G.Feitelson, 2009). The Lawrence Livermore National Lab (LLNL) workload traces were primarily chosen because the workload represents a wide range of applications with dynamic computing requirements (from large numbers of smaller to medium jobs) and with varying QoS characteristics and processing requirements. We selected a subset of the last 1,000 jobs from the LLNL Thunder workload traces from the period of

January 2007 to June 2007.

For our workloads, we synthetically generate workloads with 80% urgent jobs and 20% non-urgent jobs. Following previous work (Yeo and Buyya, 2007), we set the deadline for an urgent job to be 4 times stricter (shorter) than the deadline for a non-urgent job. For an urgent (a high-priority) job, a deadline value is assigned using a normal distribution with a mean of 1 and standard deviation of 0.2 ($\frac{1}{5}$). Alternatively, for a non-urgent (a low-priority) job, a deadline value is assigned using a normal distribution with a mean of 4 and standard deviation of 0.8 ($\frac{4}{5}$). For simplicity, we also assume the resource cost to be $uc_n = 0.1$ per time unit, and set the rental duration to $L_n = 1$ per hour. A fixed one-hour rental duration is chosen because each job from the chosen subset in the trace runs for less than 60 minutes. As described in Section 4.5.1, HASEX issues rental requests to the SGE DRM that manages a cluster of 57 nodes.

For evaluation, we focus on the following three performance metrics: mean wait time, application QoS satisfaction and resource cost to measure how well our HASEX rental-based system performs in comparison to a conventional Grid system that interconnects multiple HPC cluster systems together. As noted earlier, the mean wait time is the average time from job submission to job allocation for all job submissions to the system. Many factors will affect the mean wait time such as scheduling activity, rental decisions, and node arrivals latency. The application QoS satisfaction measures how well the system can accommodate the application's processing requirements (deadlines) throughout its execution. If too many deadlines are missed, the application QoS satisfaction will be very low. Therefore, a high application QoS denotes better performance. The application QoS is calculated as the proportion of j_s jobs whose required QoS (deadline) is satisfied out of *J* jobs submitted:

$$qs = \frac{j_s}{J} \tag{4.1}$$

We also monitor the resource cost that represents the cost involved in renting the required compute nodes in order to serve the application jobs. Based on rental or resource cost, we can measure the benefits and/or the overheads of renting extra nodes to satisfy the application QoS constraints. A lower resource cost denotes better performance. The cost for each job request *l* is computed as $c_l = R_j(uc_n * L_n)$, where R_j is the number of nodes to rent, uc_n is the resource cost per time unit, and L_n is the rental duration agreed by both the VA and the external Grid provider (SGE cluster). Hence, given total *L* number of leases initiated by the HPlanner, the total resource cost is computed as:

$$rc = \sum_{l=1}^{L} c_l \tag{4.2}$$

For this experiment, the HAllocator schedules the incoming requests in accordance with a simple FCFS (first-come first-served) priority scheduling scheme to arbitrate job requests. The HPlanner is also plugged with a simple rental policy that reacts upon a new job arrival. Upon a new job arrival, the job is placed in the queue in FCFS order. If the first job at the head of the job queue cannot be served due to an insufficient number of resource nodes, HASEX issues a rent request for the additional number

Algorithm 1 Pseudo code for resource renting.

```
1. Get the highest priority queued job $J$ from the DRM.
2. Determine the number of additional nodes $N$ that need to be deployed in order to serve job J.
3. Reserve job $J$.
4. Discover a resource provider to negotiate with (i.e., SGE DRM).
5. Issue rental request to SGE DRM.
6. Configure virtual network between local site and SGE DRM.
7. Configure new nodes with the DRM.
8. Allocate newly rented nodes to job $J$.
9. Track job state and if there are no other queued jobs, remove nodes after completion of job.
10. Repeat step 1 until all queued jobs have been processed.
```

	HASEX	Grid
Average job turn-around time [sec]	549.56	4235.67
Standard Deviation	2.924	3.123
Total Number of Job Requests	1000	1000
Inter-arrival (sec)	LLNL Thunder Trace	LLNL Thunder Trace
Average Job Length (minute)	LLNL Thunder Trace	LLNL Thunder Trace

Table 4.2: Experiment results for HASEX vs. Grid system under LLNL workload trace

of resource nodes required to serve the job. If there are no queued jobs, HASEX issues a termination request to return idle resource nodes to the DRM if the number of all free resource nodes is greater than the total number of nodes requested by all queued jobs. The pseudo code for the resource renting is summarised in Algorithm 1.

We make several assumptions as to the nature of the environment and jobs. Firstly, data-related issues, such as data safety, data transfer and network bandwidth are not taken into account. For example, the cost of transfer of a disk image for a virtual machine is assumed to be negligible, as this process is only performed once. Secondly, we also assume that all security-related issues are handled by third-party tools and that the bandwidth required for data transfers is negligible, and that it is unnecessary to use predicted bandwidth availability as a criterion for selection. In cases where high bandwidth requirements are necessary for a job execution, HASEX can query the network capabilities like bandwidth and latency between the various end points from each DRM. If network delay to the DRM is slow, then HPlanner chooses an alternative DRM route, if available. In terms of scheduling policies, we assume that there is no pre-emption and that the queue priority is based on a FCFS basis.

First, we compare the mean wait time, the application QoS, and the resource cost to demonstrate the effectiveness of HASEX, in comparison to the Grid approach. The results for the performance comparison of HASEX versus the Grid system are plotted in Figures 4.11,4.12, and 4.13. Clearly, the results demonstrate that HASEX can minimize the average wait time substantially by reducing the waiting time for queued jobs when compared to a Grid system. Without HASEX, we can observe

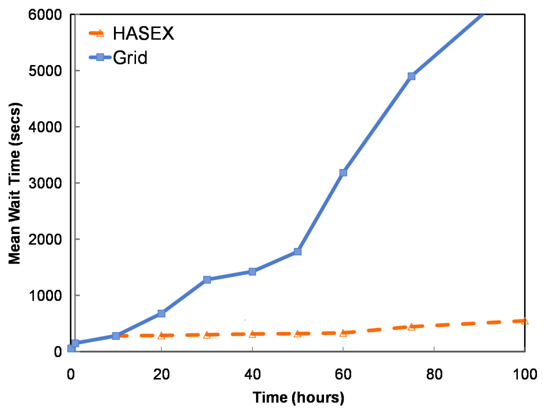


Figure 4.11: Mean wait time for HASEX vs. Grid.

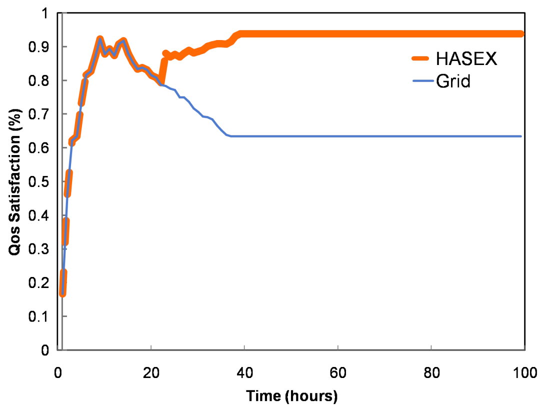


Figure 4.12: QoS satisfaction for HASEX vs. Grid.

from Table 4.2 that most large parallel jobs need to wait 15 times longer because they cannot be served immediately. As observed, when there is competition for resources, the waiting times are even higher under the Grid environment. In particular, it can be observed that the application QoS satisfaction for HASEX is relatively higher than that for the Grid system. As we can see, the HASEX system, which incorporates a rental policy, obtained an average of 87% in application QoS satisfaction in comparison to a Grid system that only achieved an average of 13% in application QoS satisfaction. However, one could doubt the effectiveness of HASEX, because of the increase in resource cost when more processors are rented to accommodate the QoS demand of queued jobs. We want to make sure that the gain that can be obtained by HASEX is significant to compensate the overhead.

Figure 4.13 shows the results of the resource cost incurred from renting additional nodes. As the competition for resource increases, more jobs are accumulated in the queue, which eventually leads to more nodes being rented by HASEX. We can observe that resource cost increases as more jobs arrive into the system (as time passes), but the cost is significantly lower in comparison to that of the Grid system. In particular, even at peak times, HASEX incurs significantly lower resource cost; it can be seen that the accumulated resource cost is reduced by a significant 47%. In fact, starting with 4 nodes initially, HASEX rents a total amount of 26 nodes at the end of experiment after all the rental negotiations. It is anticipated that the amount of total nodes could potentially be reduced further by employing a more

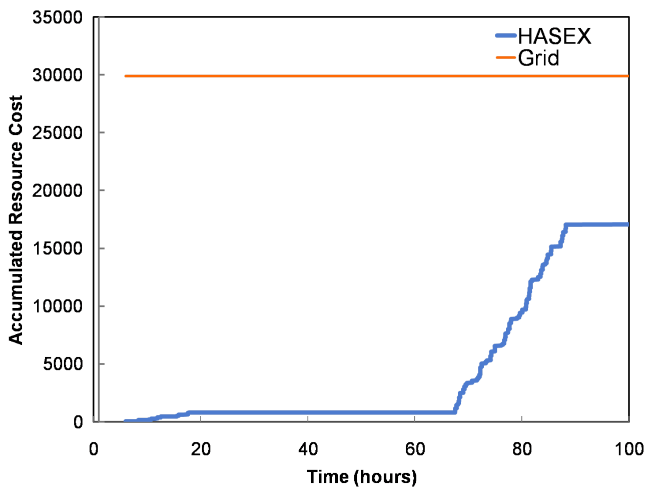


Figure 4.13: Resource cost for HASEX vs. Grid.

sophisticated rental policy.

We hypothesize that the main reason for the significant improvement observed is due to the effectiveness of HASEX in making use of the additional nodes available from the SGE DRM to serve highly urgent jobs at the most critical time. While the gLite Grid has access to a total amount of 61 nodes (i.e., 57 nodes from SGE DRM and 4 nodes from MIMOS Bhd), it is still unable to serve all jobs simultaneously. In fact, we observe that on average, 45% of the total submitted jobs at gLite are pending in the queue even though there are still many nodes available. In particular, an urgent (critical) job is forced to wait at peak time when there are multiple urgent jobs competing for nodes in parallel. In such a situation, only a few urgent jobs get access to nodes, while a significant number of jobs are being held in the queue. When that happens, the QoS of some jobs has to be compromised. This has the effect of significantly lowering the overall application QoS satisfaction. In contrast, HASEX can mitigate this issue by renting additional nodes from the external provider during peak demand periods.

The main idea of HASEX is that it can rent additional nodes when needed, and rental is dynamically adjusted against load variations. This mechanism has been shown to be effective. It can be observed that, on average, HASEX rents less than 3 additional nodes during low peak times and it rents, on average, 14 nodes during peak times. HASEX can achieve significantly higher application QoS satisfaction because more jobs are completed by their deadlines. Renting nodes significantly reduces the mean waiting times of jobs and increases the QoS satisfaction, at lower resource cost. This advantage is especially evident when jobs are highly parallel and when there is congestion in a competitive resource environment.

We obtained encouraging results that demonstrate the effectiveness of HASEX: it offers significant improvements which cannot be obtained under a conventional Grid system. In particular, HASEX has been shown to provide high performance and high QoS satisfaction for applications with significantly lower resource cost.

4.6 Discussion and Comparison to Related Work

4.6.1 On-demand Computing

Several approaches have been proposed in the literature to support the on-demand provision of computational services (Graupner, Kotov, and Trinks, 2002; Benkner et al., 2004; Irwin et al., 2006; Beckman et al., 2006; Cencerrado, Senar, and Cortés, 2009). Traditionally, these methods consist of overlaying a custom software stack on top of an existing middleware layer, like, for example, the PlanetLab Project (Peterson et al., 2006) or the VioCluster system (Ruth, McGachey, and Xu, 2005). These approaches essentially shift the scalability issues from the application to the overlaid software layer, whereas the proposed solution transparently scales both the application and the computational cluster.

The idea of a virtual cluster which dynamically adapts its size to the workload is not new. Chase et al. (2003) describe a cluster management software called COD (Cluster On Demand), which dynamically allocates servers from a common pool to multiple virtual clusters. Although the goal is similar, the approach is completely different. Cluster On Demand does not differentiate the roles of managing application processing requirements, scheduling resource requests and managing distributed resources. On the other hand, our solution is based on a multi-tier approach that differentiates the roles of application management, job/task scheduling, and resource provisioning. In our case, the solution can support a hybrid solution whereby physical nodes and virtual machines (VMs) can co-exist in the same execution environment, and the environment is built from a Grid perspective to support a wide range of parallel and high performance computing applications.

The Virtual Authority (VA) approach aligns with the concept of a multi-tier paradigm with strong reliance on dynamic provisioning via the resource renting approach. It simply constructs an environment which can be constructed temporarily from rented resource nodes. This provides the flexibility and option of storing resources ready for use under the control of a VA. As such, temporary and unexpected spikes in demand for resources can be accommodated by flexible rental arrangements. This in turn enables the applications to customize their execution with a set of distinct heterogeneous resource types and processing amounts to form ideal node configurations based on their workload requirements. As a result, a group of users and applications can share resources efficiently without the hassle of a global Grid.

With regards to Cloud computing, Eucalyptus (Nurmi et al., 2008), OpenNebula (Sotomayor et al., 2009), Tashi (Kozuch et al., 2009), and Globus Nimbus (Martinaitis, Patten, and Wendelborn, 2009) make use of virtualisation to provide on-demand clusters which are launched on Cloud systems (Sullivan, 2009). However, these systems do not provide direct support for the execution of distributed applications (Graupner, Kotov, and Trinks, 2002; Benkner et al., 2004; Irwin et al., 2006; Cencerrado, Senar, and Cortés, 2009). Although these solutions simplify the management of VMs on a distributed pool of resources, none of these initiatives for distributed VM management evaluate its extension to a grid-like environment in which different infrastructure sites could collaborate to satisfy peak or fluctuating demands.

On the other hand, HASEX provides the ability to integrate existing cluster DRM systems such as

SGE and Maui for rapid and dynamic allocation and de-allocation of resource nodes. More recently, Amazon Elastic Compute Cloud (EC2) (Amazon, 2009) has delivered cluster computing for HPC applications. Such a system provides a remote VM execution environment that allows users to execute one or more 'Amazon Machine Images' on their systems, providing a simple web service interface to manage them. Users are billed for the computing time, memory and bandwidth consumed. This service greatly complements our HASEX, offering the possibility of potentially unlimited computing resources for distributed applications. For example, HASEX can be incorporated to employ a Grid-enabled Amazon Machine Image, and create as many virtual resource nodes as needed, getting on-demand resources in cases where the physical hardware cannot satisfy a peak demand.

In summary, our approach is based on the concept of a dynamically adapting resource infrastructure which is supported via the concept of resource renting. The approach provides the ability to support both physical and virtual resource nodes – a hybrid system whereby another virtualization layer (e.g., virtual or Cloud resources) can also potentially be integrated with HASEX services. This enables HASEX to operate within cluster or Grid environments consisting of physical server nodes and/or in a Cloud environment wherein physical resources are rented out as virtual machines. This effectively introduces the possibility for parallel and HPC execution in Cluster, Grid and Cloud environments. Given this capability, it is envisaged that HASEX can become a uniform interface to all computational resources for high performance computing needs.

4.6.2 Economic-based Resource management system

Sutherland (1968) was a pioneer who proposed economic-based allocation on PDP-1 (Levy, 1984) computers over forty years ago. Irwin, Grit, and Chase (2004) expanded this model to the Grid and pushed the idea of a virtual currency, or 'Grid credits', allocated by an independent governing entity. Recent examples of such systems include Spawn (Waldspurger et al., 1992) and REXEC (Chun and Culler, 2000). Spawn (Waldspurger et al., 1992) is an economic-based computational system that runs on a network (mostly LAN) of idle heterogeneous high-performance workstations running Unix. Spawn employs a Vickrey, sealed-bid, second price, auction mechanism. 'Sealed' refers to the notion that bidding agents have no access to other agents' bid information and 'second-price' means that the highest bidder will pay the price offered by the second-highest bidder. In such a system, if there is a unique bid for a resource, the bidder gets it for free, in the absence of competition. REXEC (Chun and Culler, 2000) is a batch job scheduler for clusters of workstations. REXEC users assign tickets to their jobs. While executing, the REXEC accounting system charges users proportionally to their job tickets over the sum of all executing jobs tickets.

Developed at HP Labs, Tycoon (Lai et al., 2004) is also another economic-based distributed resource allocation system based on proportional share allocation. Along with other economic-driven schedulers, it allows users to differentiate the value of their jobs in order to differentiate urgent and non-urgent jobs. Tycoon implements continuous bids where users express a bid for a resource on a given processor at a given price point for a duration. On each processor, the auctioneer calculates each bid and allocates resources in proportion to bids. While this continuous bidding scheme alleviates the user's burden of posting bids for each job, it does not guarantee resource allocation prior to execution which is a major impediment to most users with computational deadlines (Yeo and Buyya, 2007).

SHARP (Fu et al., 2003) is an architecture for secure resource peering and sharing across participants and trust domains. Its stands for Secure Highly Available Resource Peering and is based around timed claims that expire after a specified period. SHARP implements tickets that are a form of soft claim that does not guarantee resource availability for the specified timeframe. Tickets enforce probabilistic resource allocation in a manner similar to an airplane ticket reservation that only becomes 'real' once the boarding pass is printed. This allows SHARP system managers to overbook resources and to defer hard claims or rental allocation until execution. SHARP also presents a very strong security model to exchange claims between agents, either site agents, user agents or third party brokers, which achieves identification, non-repudiation, and encryption.

Mirage (Chun et al., 2005) is based on a microeconomic resource allocation system based on repeated combinatorial auctions (a form of double auction expressing tradeoffs between various options) for a test-bed of 148 nodes wireless sensors. Furthermore, one of the most interesting works investigating the realms of a true Grid Economy is the G-commerce project (Wolski et al., 2001b). The project considers an unregulated economic system where consumers and providers act in their own interest for the system to eventually reach an equilibrium.

Buyya (2002) and Sherwani et al. (2002) examine the use of supply and demand-based economic models for the purpose of pricing and allocating resources to the consumers of Grid services. They assume a supply and demand-based economy in which both software services and computational resources are in demand. For example, Sherwani et al. (2002) proposed hard QoS support which guarantees that accepted jobs are finished within their deadline QoS. Buyya (2002) also offers service level agreements control whereby failure to deliver the agreed level of service can result in penalties that would reduce the financial benefits of the resource providers.

While the above mentioned works employ economic mechanism in their systems, our work extends beyond the scheduling level. The multi-tier paradigm proposed within the VA separates the scheduling tier from the resource provisioning layer due to the core of our resource renting approach. The resource provisioning tier is acted upon by the resource renting mechanism which is then enforced by the VA. To the best of our knowledge, the work is the first attempt that examines the economic aspect of managing distributed resources under a multi-tier controlled environment where both the users and resource providers could potentially operate with different rental policies and varying cost options.

4.6.3 Decentralized Scheduling and Resource Discovery

A number of research efforts have recognized the need for decentralized Grid scheduling algorithms. In Condor-G (Frey et al., 2002), 'flocking' allows a Condor scheduler to submit allocation requests to another cluster when no resources are available locally. However, it requires schedulers to have direct connectivity with all the individual machines of every cluster it wishes to communicate with.

SmartGRID (Huang et al., 2008) propose a decentralized and interoperable Grid scheduling framework that integrates the local infrastructure information provided by swarm intelligence technology to construct a consensual Grid community. Swarm intelligence (Eberhart, Shi, and Kennedy, 2001) is a branch of artificial intelligence that focuses on distributed collaborative algorithms inspired by the behaviour of swarms of insects. Bioinspired solutions have already been successfully applied to several network routing problems (Schoonderwoerd et al., 1996; Di Caro and Dorigo, 1998), as well as to resource discovery in unstructured networks (Michlmayr, 2006).

More recently, there have been several published algorithms (Drost, Nieuwpoort, and Bal, 2006; Iordache et al., 2007a; Fiscato, Costa, and Pierre, 2008; Costa et al., 2009) that offer scalability properties for large-scale resource discovery. For example, Drost, Nieuwpoort, and Bal (2006) propose dissemination algorithms to locate a number of idle machines upon a job request. The dissemination infrastructure explores close-by nodes first so that the resulting set of machines exhibit some degree of geographical locality.

Chakravarti, Baumgartner, and Lauria (2004) propose mechanisms that allow applications be in control of the allocation of resources to them. Although this approach potentially allows fine tuning of the resource selection to the precise needs of individual applications, it forces the application to perform resource discovery rather than focusing on scheduling to maximize application-related performance. Furthermore, Iordache et al. (2007b) propose peer-to-peer-based genetic algorithms to build efficient schedules in an open environment. This approach has been shown to have very good performance in small-scale environments, but it relies on a global knowledge about the condition of the large-scale environment. More recently, Fiscato, Costa, and Pierre (2008) further extend similar genetic algorithms that only require very partial knowledge about the large-scale environment as a whole. The above mentioned works are undoubtedly interesting aspects that can be exploited to provide the rapid resource discovery mechanism within HASEX.

4.7 Chapter Summary

In this chapter, we have described the HASEX system that provides the mechanisms whereby resource nodes can be aggregated across sites in a dynamic and incremental fashion. In particular, we have focused on the design and implementation of general, extensible abstractions for HASEX services.

A conventional Grid system typically performs resource allocation according to top-down policies using a meta-scheduler (Grid scheduler) that optimizes the system-centric performance metrics from the available clusters of participating sites (Yeo and Buyya, 2007). The HASEX system, on the other hand, is based on a multi-tier approach that differentiates the roles of application management, job/task scheduling, and resource provisioning. It significantly reduces the overhead of managing resources via a renting approach. It allows one to build a virtual authority (VA) from whatever hardware configuration best suits the application, while providing the option to rent additional resource nodes to cope with future fluctuations in demand. Since HASEX decouples job scheduling and resource allocation policies from resource provisioning (rental) policies, the system can accommodate application requirements with varying resource needs at lower cost.

The observations and preliminary results provide early evidence supporting the viability of the multi-tier VA approach via the HASEX prototype system. Despite this success, one of the key difficulties we had in performing these experiments was creating a meaningful workload. In particular, obtaining a meaningful application workload that could drive the system to specific conditions was difficult. There were simply too many factors that needed to be controlled and taken into account, such as bandwidth variation and resource availability, etc. Moreover, software that implements the prototype and data collections need to be tested rigorously for both stability and correctness in order to ensure the credibility of the results. Therefore, performing a meaningful rental-based experiment on a significant scale is a challenging task.

To further evaluate the system's applicability and its potential, we turned to simulation. While it is definitely worthwhile to perform empirical studies, simulations are also valuable, since they allow systematic explorations of performance under a wide variety of workloads. Towards this end, the experiments conducted for the remainder of this thesis will be based on the results from an extensive series of simulations.

Chapter 5

Cost-Benefit Analysis of Resource Management Infrastructures

5.1 Introduction

The success of managing distributed resources at a national scale does not lie only in minimizing waiting and response times. Regardless of the underlying platform (i.e., clusters, supercomputers, desktops), users want to be able to execute their applications without making a huge investment in new hardware. Resource sharing is about leveraging the available resources and idle processor cycles to more quickly solve a problem while at the same time maximizing efficiency and reducing the total cost of ownership.

To justify the total cost of operating resources, the resource management system needs to consider the revenue that the system will gain by serving the applications' QoS requirements against the cost to operate resource nodes for satisfying such QoS requirements. However, adoption of distributed computing platforms and services by the scientific community is in its infancy as the performance and monetary cost-benefits for scientific applications are not perfectly clear. This is especially true for conventional Cluster and Grid systems which have been widely deployed for the purposes for provisioning data and compute resources. Resource sharing of geographically distributed resources is normally controlled by a global scheduler (also known as a meta-scheduler) that manages local schedulers (distributed resource managers) across geographically disparate participating sites worldwide. The meta-scheduler disseminates and manages resource information from potentially hundreds of thousands of local schedulers worldwide. A single centralized approach seems very promising from an overall viewpoint.

Unfortunately, as noted in previous chapters, the efficiency and scalability problems make such a solution appear too cumbersome for individual users. It may therefore be more efficient and cheaper for an individual user to build a small private resource system to serve his/her high performance computing (HPC) needs. However, a small private system may not be powerful enough to accommodate the high requirements of scientific applications that have complex computation workloads with sudden surges in demand. Alternatively, users may choose to purchase and maintain a large dedicated resource management system, in order to meet the demand for high performance computing and to have sufficient control over its use. This option, however, could be far more expensive.

To investigate this issue, we perform a cost-benefit analysis of a small private resource system consisting of 32 nodes for a total of 128 processors and a large dedicated HPC system with 1,024 nodes (a total of 4,096 processors). Using real cost data collected from both the LUNAR project (Sumby, 2006) and the Lawrence Livermore National Lab (LLNL) (Feitelson and Rudolph, 1995) computing facility, we carry out a cost comparison of the small private system with Grid system, which is currently the standard approach for large-scale distributed resource sharing.

5.2 Computing Costs for a Small Private Resource System and a Large Dedicated HPC System

Before we can make useful evaluations, it is important to estimate the cost of purchasing and maintaining a private resource system and a large dedicated HPC system. A more correct and measurable number should be based on a financial model, where the cost estimate includes all hidden costs, such as infrastructure and maintenance costs. *How much does it really cost to operate a private resource and a dedicated HPC system to serve jobs from individual users and applications?*

To help answer this question, we use financial expenses, collected from several small and large HPC systems (Koomey, 2007; D.G.Feitelson, 2009) to perform TCO (total cost of ownership) analysis, with an emphasis on systems that are constructed to support a wide range of HPC users and applications. With this data, we use 'back-of-the-envelope' calculations, based on several factors, such as the number of CPUs, the initial hardware cost (which includes the mainboard, storage and racks), the personnel/system administrator costs (salaries for software and hardware maintenance), business premises, and electricity costs (which includes the additional power requirements for cooling and power delivery inefficiencies). A TCO analysis would help organizations evaluate what they actually pay to operate their systems in terms of hard costs for hardware, software, services, as well as costs such as personnel time, downtime, maintenance, upgrades and enhancements.

We detail the costs of maintaining a short HPC computing project. Table 5.1 summarizes the cost breakdown for a relatively small project called LUNAR using approximately 128 processors (32 nodes). LUNAR is an astrophysical project with an objective to map the temperature of the lunar surface over a specified period in order to find the existence of ice on or underneath the moon's surface. The project runs only for a very short duration i.e., five months, but it requires significant resource requirements. The project involves complicated calculations that need to be solved within real-time deadlines, in which the processing requirements required to solve the calculations are highly dynamic, containing unpredictable demand spikes.

As can be observed from Table 5.1, the principal cost is that of the system administrators'/programmers' salaries, constituting almost 32% of the TCO. The network connection has the second highest share of the TCO; approximately €5,192 per month. The business premises (to accommodate floor space) component constitutes the third share of the TCO while the amortised cost of servers constitutes the fourth share of the TCO. The cost of electricity for power and cooling infrastructure contributes the lowest TCO. The total monthly costs are approximately €17,219 per month.

	Number of CPUs	128 (32 nodes)
Cost Factor	Monthly (€)	Total (€)
Equipment cost amortised over the 3-year lifetime	2,489	12,444
Personnel Cost (Salaries)	5,667	28,333
Business Premises	2,667	13,333
Network Connection	5,192	25,958
Electricity (inc. cooling power)	1,205	6,027
	Total Cost	86,096
	Total Monthly Cost	17,219

Table 5.1: Estimated project cost (total cost of ownership) for a five-month period. The LUNAR project runs are very short, (i.e., five months), but have significant resource requirements (with a total of 128 processors used during the project).

Likewise, we also compute the costs for the Lawrence Livermore National Lab (LLNL) (D.G.Feitelson, 2009) – a large institution that operates over the long term with five years of operation. Again, using back-of-the-envelope calculations, we derived the total cost for the construction and maintenance of the LLNL system. Table 5.2 shows the full cost breakdown for construction and maintenance of a dedicated HPC LLNL system with 1,024 nodes (4,096 CPUs). The equipment costs are written off after the completion of three years of operation, and additional costs are also included for upgrades and maintenance from vendors. The equipment cost is incurred when purchasing hardware during the start-up phase. However, after hardware costs are amortized over a period of three years, the amortized cost of servers comes to €58,287 per month. By amortising, we obtain a common cost run rate metric that we can apply to both one time purchases (e.g., for nodes/processors) and ongoing expenses (e.g., elecricity, floor space, staff etc.).

	Number of CPUs	4,096 (1024 nodes)
Cost Factor	Monthly (€)	Total (€)
Total Equipment Cost, i.e., CPUs over a 5-year lifetime (inc. upgrades)	58,287	3,497,200
Personnel cost (salaries)	79,333	4,760,000
Business Premises (floor space)	47,046	2,822,740
Network Connection	23,783	1,427,000
Electricity (inc. cooling power)	47,046	2,822,740
	Total Cost	15,329,680
	Total Monthly Cost	255,495

Table 5.2: Estimated total cost of ownership for LLNL with 4,096 processors for a 5-year period.

It can be seen that the highest cost component is attributed to personnel costs (staff salaries) which make up 30 percent of the TCO. Personnel costs at LLNL include salaries for skilled staff members, namely, six system administrators and eight system developers. The cost also includes additional costs for hardware and software installation, maintenance, and upgrade services by vendors. A ratio of approximately 130 nodes per personnel staff/system administrator is assumed (*Server Support Staffing Ratios* 2006). The expenses for full-time staff are also highly dependent on their respective countries, due to large differences in wage levels. We focus the situation within Europe[1], where the average cost to employ a system administrator is approximately €68,000 per annum (which includes government taxes and social insurance contributions) (Opitz, König, and Szamlewska, 2008). The business premises, including floor space for the mount racks, and equipment cost €230 per month. The calculation is based on the assumption of a 4 x 8 foot (32 square foot) area to support 6000 pounds of rack-mounted hardware. For a smaller number of nodes, the network cost is lower since the LAN costs are neglected as LAN transports typically cost 1000 times less than WAN transports (Opitz, König, and Szamlewska, 2008). Comparing the total cost per month of both the LUNAR project and the LLNL HPC system, it can be observed that a central processing centre for a large corporation (e.g., LLNL) is even more expensive to build and maintain than a small private resource system such as that constructed for the LUNAR project.

This demonstrates that it is much cheaper to build a small private system in comparison to building a large dedicated HPC system. Building and maintaining a large dedicated HPC system is simply not cost effective, especially when users do not run jobs day and night continuously for the whole three-year period. Rather, it is more cost effective to build and maintain a small private resource system that provides continuous usage of equipment during a project's lifetime, but such a system should also be equipped with the ability to acquire extra resources when they are needed for sudden peaks in demand. This would meet the high response-time requirements for users with very large computations and the sudden surges in demand. It is anticipated that such additional resources could be obtained temporarily (at sudden peak in demand) from under-utilized production Grid environments. In particular, Grid computing offers potential access to parallel CPU capacity due to the high availability of worldwide resources, which also provides a huge opportunity for exploiting under-utilized resources[2].

5.3 Computing Costs for Grid Computing

Next, we estimate the real cost of building and maintaining production Grid systems. This is important for us to evaluate whether there is any real cost-benefit in tapping the global pool of unused Grid resources versus the cost of building a dedicated HPC system. For comparison, we chose the Enabling Grids for E-SciencE (EGEE) as a benchmark to estimate the true cost of building and maintaining real Grid systems. The EGEE was chosen because it is currently the largest production Grid system. Furthermore, the cost breakdown data on the Enabling Grids for the E-SciencE (EGEE) project is also available online for reference (Opitz, König, and Szamlewska, 2008).

Table 5.3 summarizes the cost breakdown to build a Grid system. We can see that the total monthly cost to run a production Grid is €18,870,000. Based on this, we can observe that the total monthly cost to run EGEE production Grids is significantly higher in comparison to a private resource system or a large dedicated HPC system. If we break down the cost even further, we obtain the cost of €0.26 per Grid node per hour.

It is anticipated that if the real cost to acquire a Grid node is indeed €0.26 per node per hour, then there is no reason why the nodes should not be available at this cost for use by a private resource system.

[1]The premises space costs, salaries and even equipment can vary greatly across different parts of the world.

[2]Most Grid production sites, such as; DAS-2, Grid500, NorduGrid, AuverGrid, SHarCNET, and LCG, have a system utilization of less than 60%. In some cases, the system utilization is well below 40% (*The Grid Workloads Archive*).

Number of CPUs	100,945

Cost Factor	Total (€)
Annual Hardware	176,653,750
Annual Personnel Cost (salaries)	19,200,000
Annual Business Premises	25,236,250
Annual Network Connection	5,350,000
Annual Electricity (inc. cooling)	4,000,000
Total Yearly Cost	226,440,000
Total Monthly Cost	18,870,000
CPU Cost per Day	6.15
CPU Cost per Hour	0.26

Table 5.3: Estimated cost for EGEE (Enabling Grids for E-SciencE) with a total number of 100,945 CPUs across 319 international sites.

It is anticipated that if we can acquire a node at this cost from the EGEE Grid, we can effectively make the best use of the current widely available Grid resources to serve users, without the need for them to purchase or maintain a dedicated large HPC system. In particular, a hybrid approach is advocated where the small size of a private resource system is retained, but the system has the flexibility to increase or reduce its size in order to fit certain user demands.

5.4 Virtual Authority

Having established the real costs for a private resource system, for a large dedicated HPC system and for Grid computing, the burning question is this: *"How can the system be made profitable for both lender and borrower?"* The lender that we refer to here is the Grid resource provider that rents out some of its resources (nodes) to borrowers i.e., resource consumers, which are referred to as users and applications. As noted in previous chapters, a Virtual Authority (VA) is constructed from a customized set of resource machines that fit certain user demands. A VA is a collection of resources controlled, but not owned, by a group of users or an authority representing a group of users. When a resource is rented out, the owner of a resource recognizes only the VA. All permissions, billing and blame for security breaches are associated with the VA.

A VA system is simply a third party entity between users and resource providers: it receives user demands and at the same time manages its own resources and rents further additional resources if current resources are insufficient to meet the demands. Without a VA system, the only alternative for consumers may be to purchase and maintain their own systems, which could be far more expensive. It is anticipated that having the option to outsource workloads could in fact lower the cost for the consumers compared to building and maintaining their own nodes. Since such a VA system can be tailored accordingly, runtime negotiations for nodes are promoted. This gives users the opportunity to rent and release nodes based on the fluctuations of load demands. Neither consumers nor users are obligated to pay for resources when they are no longer required. This provides the possibility to utilize less attractive resources at specific stages. From the perspective of a VA system, it must be able to handle a variable demand load and, at the same time, justify the investment of acquiring external Grid resources. The question would then be: *"How much will it cost for the system to handle user and application demand satisfactorily?"* In the remainder of this chapter, we will examine and answer this important question.

5.5 Evaluation Methodology

To capture the dynamic behaviour of the environment, our experiments use a workload that is drawn from a characterization of a real supercomputing environment at Lawrence Livermore National Labs (LLNL) from the Parallel Workloads Archive (D.G.Feitelson, 2009). Job arrival, execution time and size, number of CPUs – hereinafter referred to as 'tasks information' – have been traced from the workload. We have constructed a simulated environment to mimic the large LLNL HPC system using the available workload traces and have used it to evaluate the cost and the performance against that of the VA approach.

We have chosen in this chapter to evaluate our architecture using a simulation framework designed for this purpose. There are two reasons for doing so: firstly, the impact of policy changes will typically become measurable over the course of several days rather than immediately, making experiments long and difficult to repeat in real environments. For example, the average runtime for each job can be up to one hour, implying that it would take several days to run 1,000 jobs before the impact of users' submissions become noticeable for evaluation. Secondly, whilst it is necessary to rely on diverse workloads to evaluate the allocation process, it is also necessary to retain tight control over the parameters of the workload distributions, such as the overall mean and variance of job lengths, in order to verify fair-share-type scenarios. The simulation framework will be documented in detail in Appendix A, but its capabilities are briefly described in this section.

The high-level architecture of the simulation framework is illustrated in Figure 5.1. The framework is based on CLOWN (Sorensen and Jones, 1992) , a discrete event-based simulation framework in C and C++, designed to simulate queuing networks. CLOWN provides the necessary constructs to allow us to model independent entities as parallel processes and handles the required synchronization and event scheduling. The framework is highly flexible and allows us to simulate anything from job queues and allocation mechanisms to task processing as well as providing statistical and reporting support.

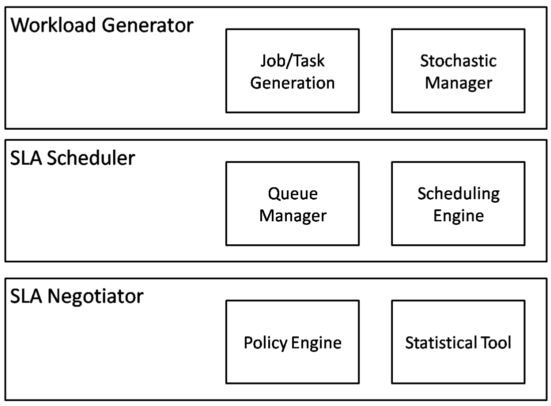


Figure 5.1: Architecture for extension of the CLOWN simulation framework.

We have extended CLOWN to mirror the system architecture described in Chapter 3: the Application Agent, the VA Scheduler tier, the VA Negotiator tier, the Resource Providers tier, and the underlying resource nodes. We have also extended HASEX to model our performance metrics and the investigated policies. A wide range of models and policies can be simulated using CLOWN, by simply implementing new algorithms and redefining the relationships between entities.

The extended CLOWN simulator consists of four core modules: Workload Generator, VA Scheduler, VA Negotiator and Resource Providers. The application agent is represented by the Workload Generator module. We do not explicitly implement individual application agent behaviours. Instead, the Workload Generator consists of a sequence of jobs that are submitted for execution by several users at arbitrary times based on the chosen workload distribution. This scenario is often referred to as being 'online', namely, that the submission times of jobs (also called their 'arrival times') are a-priori unknown (Tsafrir, Etsion, and Feitelson, 2007). Each job is characterized first and foremost by its arrival time, execution time and job size (number of CPUs).

The system components in CLOWN are represented using modules, e.g., the Workload Generator, VA Scheduler, VA Negotiator, and Resource Providers modules. The modules interact with each other primarily through two types of events flowing over module-to-module interconnections: job events and feedback events. Job events model the flows of jobs from the Workload Generator which generates them down to the nodes that execute them. The Workload Generator contains the Job/Task Generation and Stochastic Manager sub-modules that are used to generate job events which are passed to the VA Scheduler module. The VA Scheduler consists of the Queue Manager sub-module that represents the first point of entry in which the job events are queued to be scheduled. The VA Scheduler also consists of the Scheduling Engine sub-module that implements the scheduling policy for these jobs.

Further, the VA Negotiator consists of the Policy Engine module that performs the necessary rental activities based on the states and events from the VA Scheduler. When a rent condition has been triggered, it sends a rental event to the Resource Providers module, and the Resource Providers module then sends a feedback event to pass resource information about the rented nodes. The rented nodes are then attached under the control of VA Scheduler and VA Negotiator upon receipt of this notification. The VA Negotiator also generates a suitable analysis of the simulation using the Statistical Tool sub-module created for this purpose.

In summary, our simulations are driven by simulation modules which generate new events and provide module-specific event handling code. The events are controlled by a simulation engine which continuously processes the earliest event in virtual time across all modules until a stopping condition is reached. From the point of view of the engine, the modules (Workload Generator, VA Scheduler, VA Negotiator, Resource Providers, and Resource Nodes) are simply black boxes that contain event queues and have associated event handling code.

5.6 Evaluation

The workload traces of the LLNL Thunder system contains over 100,000 jobs, which are based on the trace period from 1 Feb to 30 June. We select a subset of the first 10,000 jobs[3] from the trace as input

[3]The amount of jobs has been determined after several experimentations with different quantities of jobs, starting from 100 jobs to 100,000 jobs. As a result, 10,000 jobs was chosen as a suitably significant amount to use to evaluate the system. The choice has also been corroborated by other related work. Irwin, Grit, and Chase (2004) uses a trace of 5,000 jobs to represent workload characteristics. Alternatively, Popovici and Wilkes (2005) present different experiments using 200 jobs only.

for the event-driven simulation, and the evaluation of the various rental policies is performed by using a discrete-event simulator. From our analysis, the workload contains an average of 42 processors, average inter arrival time of 107 seconds (1.8 minutes), and average runtime of 2186 seconds (36.4 minutes). These LLNL workload traces were chosen because the system was used by a large number of users (283 users) with different HPC computing requirements (from large numbers of smaller to medium-sized jobs). The LLNL workload is also currently the most recent production system workload available from the Feitelson Parallel Workloads Archive (D.G.Feitelson, 2009). Hence, it captures and represents the latest nature of a real workload and a real system for a HPC system.

Our rental policies consist of several important functions. The most important are the *rent* and *release* functions, which determine whether to rent additional nodes or to release current rented nodes, respectively. Our algorithm also responds to a number of events, which include job arrival (when a new job request arrives to the system), task completion and job termination (when a task or a job has completed its execution). The pseudo code of the scheduling and rental policies is presented in Figure 2.

When a new job arrives, the job is placed in the queue to be served on a first-come-first-served (FCFS) basis (lines 4-7). The system looks in the queue to serve the next job until all jobs are completed or when the nodes are insufficient to serve the job (lines 8-21). If there is a sufficient number of resource nodes that can fulfil the specified resource requirements of the job, the rental event is increased by one (lines 22-24). Subsequently, if the rental event has gone beyond the rental threshold, the system initiates a rental request for the remaining number of additional nodes for the job (lines 25-40). For each task completion, the node held by the task is freed and subsequently the release event is increased by one (lines 41-43). If the number of release events has gone beyond its release threshold, the system will react by initiating a request to relinquish idled rented node(s) (lines 44-47). Next, the system finds another job to serve in the queue if there is any (lines 48-50). Finally, at the end of each job completion, the performance statistics are updated (lines 52-54).

In this algorithm, both the rent and release policies are state dependent, whereby after specific events have occurred, the VA Negotiator determines the number of additional nodes to rent based on (i) the total number of queued job requests and (ii) the current number of rented nodes. Similarly, the system also determines how many rented nodes to release based on the same parameters.

5.6.1 Performance Metrics

The current EGEE has over 100,945 CPUs at 319 international sites, which are constantly available to users for use. Given the high availability of CPU resources, it can be safely assumed that the CPUs can be reserved and rented out at any point in time. A node is rented based on its rental duration or reservation. A lease is an agreement between the service provider and the Grid resource provider to rent nodes within a specified duration. Leases are non-pre-emptable, which means that once a node is rented, the Grid provider cannot request back the node until the lease expires. However, a VA may wish to release the node it reserves before the lease expires. When a VA makes an order, the time it takes for the nodes to arrive is exponentially distributed with an average of 10 minutes (Iosup et al., 2006). The VA earns revenue based on the consumption charge η_n and the CPU usage for total *N* nodes in hours:

Algorithm 2 Pseudo code for base rent and release policies.

```
1:  Rent events: $RE = 0$
2:  Release events: $DE = 0$
3:  Rental Threshold events: $\Omega = \{1, 10\}$
4:  if Job's arrival event is triggered then
5:      Insert new arriving job into the FCFS job queue
6:      Invoke the SCHEDULE function.
7:  end if
8:  if SCHEDULE function is invoked then
9:      FCFS queue = $Q$
10:     Number of jobs in Q = $Q.size$
11:     if $Q$ is empty then
12:         return
13:     end if
14:     Head of $Q$ = Job $J$
15:     $J.size$ = number of nodes required by $J$
16:     $F.size$ = the current system capacity
17:     if $J.size$ = C then
18:         Update $F.size = F.size - J.size$
19:         Remove $J$ from $Q$
20:         Start executing J
21:     end if
22:     if $Q.size > 0$ and $F.size < J.size$ then
23:         Increase $RE$ by 1
24:     end if
25:     if $RE \geq \Omega$ then
26:         Reset $RE$ to 0
27:         Invoke the RENT function
28:     end if
29:     if $Q.size > 0$ and $J.size > N.size$ then
30:         Invoke the RENT function.
31:     end if
32: end if
33: if RENT function is invoked then
33:     $TP$ = total number of nodes requested by $J.size$.
33:     $RN$ = number of nodes to rent.
34:     if $TP > F.size$ then
34:         Rent $RN = TP - F.size$
34:         Waits for $RN$ nodes to arrive
35:     end if
36: end if
37: if RELEASE function is invoked then
38:     if $F.size > TP$ then
38:         Release $RL = F.size - TP$
39:     end if
40: end if
41: if Task's Completion event is triggered then
42:     Increase the $F.size$ capacity by 1
43:     Increase $DE$ by 1
44:     if $DE \geq \Omega$ then
45:         Reset $DE$ to 0
46:         Invoke the RELEASE function.
47:     end if
48:     if $J.size > 0$ then
49:         Invoke the SCHEDULE function.
50:     end if
51: end if
52: if Job's Completion event is triggered then
53:     Update statistics on cost and performance
54: end if
```

$$network = \sum_{n=1}^{N} \zeta_n \tag{5.1}$$

and

$$lease = \sum_{n=1}^{N} \eta_n (t_r - t_c) \tag{5.2}$$

where ζ_n is a network fee and η_n is a consumption charge for node *n,* which is a fee paid per time increment. Based on previous calculations in Section 5.2, the consumption charge η_n for acquiring a Grid node is €0.26 per hour. The lease or resource fee is charged from the moment the system takes control of the node (t_c) until the node is returned to the Grid resource provider or its owner (t_r). The consumption η_n cost remains constant for the duration of the lease agreement, but the rental component η_n may well be influenced by the utility making popular nodes more or less expensive. The administrative costs represent the network costs for WAN interconnects for having to deploy and configure a Grid node from an external site. Based on Opitz, König, and Szamlewska (2008), the network cost incurred for deployment, communication etc. costs €0.09 per CPU[4] and is charged only once for the entire lease period, assuming that 1Gbit network links are used. We also assume that resources are always available during times of high demand from Grid providers. We realize that this is a big assumption, but it is a realistic one given the high availability of resources from today's Grid systems.

Ideally, there must be a cost associated with job execution. The standard measure that we use is response time. The response time is the difference between the arrival of the job and when the job has completed its execution. The system's goal in this case is to minimize response time, which can be affected by two factors: (1) waiting time: how long a job has to wait before it can execute and (2) node processing speed: how soon the job can be completed with the processing power of nodes allocated to it. The mean response time (rpt) is the average response times for all jobs. The mean response time (rpt) will be used as one of the performance metrics for evaluation. Furthermore, using our performance metrics, we intend to capture both network and lease cost using a single aggregate cost metric to simplify the cost-benefit comparisons of the different rental policies:

$$cost = network + lease \tag{5.3}$$

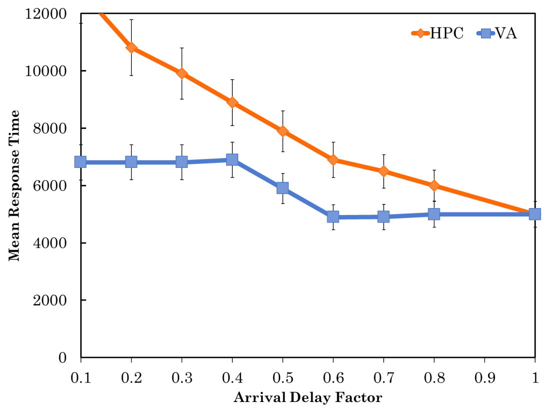
The lower the cost attained by the VA system, the less the system can charge users for executing their jobs. Hence, the advantage of the cost metric is that it captures the overall effectiveness of the investigated policies with just a single metric.

5.6.2 Baseline Performance

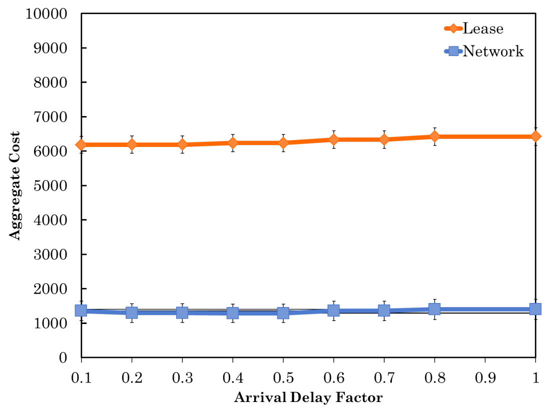
In this experiment we perform a baseline comparison between a dedicated HPC system and a VA. Does the VA approach really improve system performance, i.e., the response time when compared with a dedicated HPC system, and how does this affect cost? How much savings can one make to have a

[4]The cost for downloading large data files from remote sites is not considered in our evaluation.

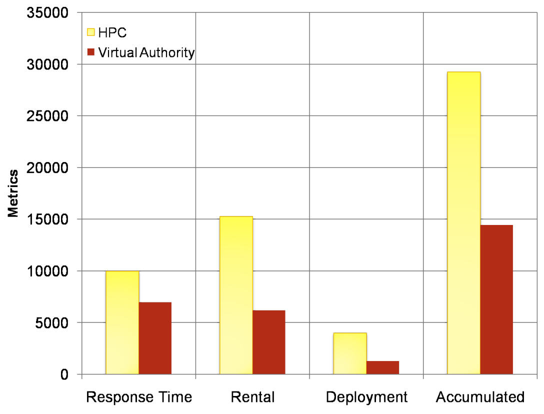
similar performance to that of a dedicated HPC system? We aim to show the ability of the VA approach to automatically rent the optimal number of nodes to maintain system performance for the execution of jobs. The resulting cost breakdown of a dedicated HPC system versus a VA approach is presented in Figure 5.2.



(a) Mean response time.



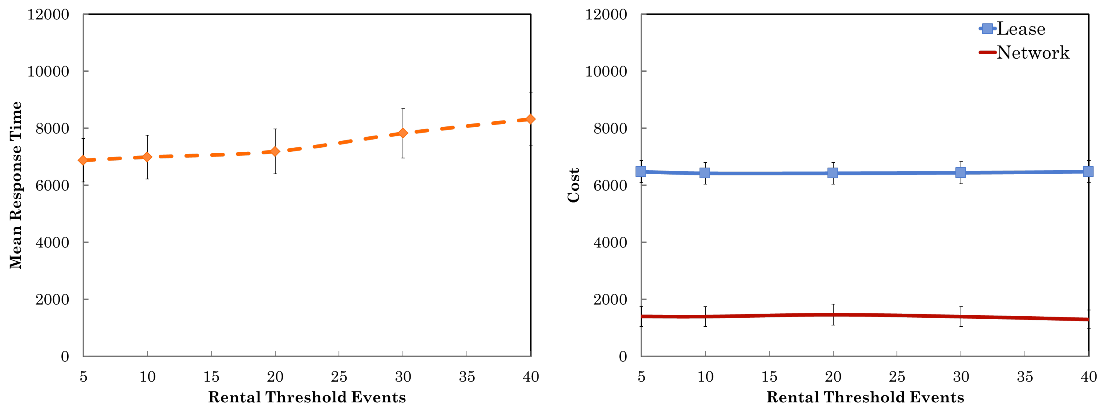
(b) Average cost.



(c) Cost breakdown.

Figure 5.2: Cost breakdown for a VA system and a dedicated HPC system. The rent and release policies are presented in Algorithm 2.

Figure 5.2 shows the cost breakdown and how a VA performs, with respect to the dedicated HPC system, for a changing workload. We model varying workloads through the arrival delay factor, which sets the arrival delay of jobs based on the inter arrival time from the trace. Hence, a lower delay factor represents a higher workload by shortening the inter arrival time of jobs. Interestingly, we observe that the mean response time, the network, and lease costs do not change greatly as the load increases (when the arrival delay factor decreases). This shows that the cost to operate a VA system is not greatly affected by increasing loads. Moreover, the VA system operates at a lower cost compared to a dedicated HPC system and it gives better overall performance. In contrast, the mean response time, the network, and lease costs are fairly high for a dedicated HPC system. This is due to the fact that the number of nodes in the dedicated HPC system is static, resulting in a fixed lease overhead. Therefore, at sudden increases in demand for resources, the mean response time increases when the load is increased. On the other hand, a VA system is able to mitigate this effect by renting additional nodes as it adapts to load changes. As a result, we observe that the mean response time, the network, and the lease cost differences for a VA system are negligible as the load increases.



(a) Mean response time. (b) Cost.

Figure 5.3: Cost breakdown for VA under increasing rental threshold events.

Figure 5.3 shows the breakdown of mean response time and cost, as a function of rental threshold events (Ω). We observe that the cost differences for the network and the lease costs are negligible, as the rental threshold events increase. This indicates that both network and lease costs are not sensitive to the rental threshold events. For mean response time, we see a slight increase as the rental threshold events increase. This is due to the fact that higher rental threshold events force queued jobs to wait longer. This results in slightly higher waiting times before the next leasing decision is triggered. This observation shows the effectiveness of the VA to perform immediate rent and release actions by taking into account of the mean response time at each scheduling period, before the mean response time gets too high.

5.6.3 Impact of Scheduling Schemes

So far, the scheduling scheme in use has been based on First Come First Served (FCFS). *How does a VA system perform under any other scheduling scheme?* We aim to show the cost differences under different scheduling schemes. Figure 5.4 compares the FCFS scheduling scheme with other well-known scheduling policies, such as Shortest Job First (SJF), Longest Job First (LJF), and Shortest Task First (STF). SJF schedules jobs in ascending size, from the smallest to the largest job. LJF schedules jobs in descending order, from the largest to the smallest jobs. STF schedules jobs in ascending order of job execution time, assuming that prior knowledge of task execution time is known. We observe that all scheduling schemes display a comparable cost. Out of all schemes, STF incurs slightly lower aggregate costs when compared to other scheduling schemes. FCFS gives the second best cost. Although STF is a clear winner, the scheme assumes perfect knowledge of job execution time (CPU time), which is not often realistic for interactive or dynamic jobs. Therefore, we restrict our discussion to a FCFS scheduling scheme for the rest of the evaluation.

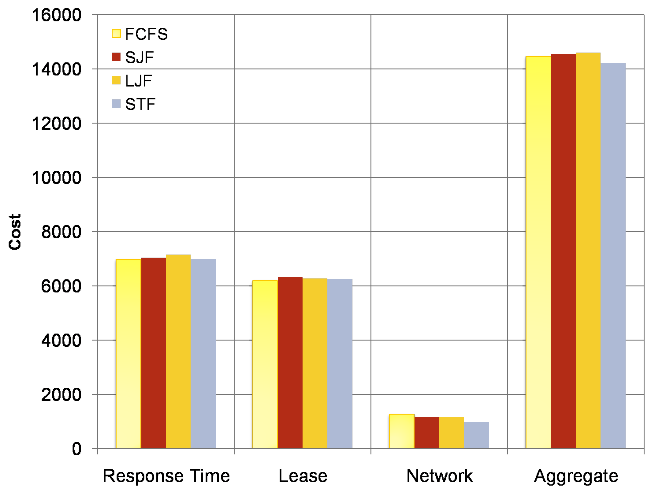
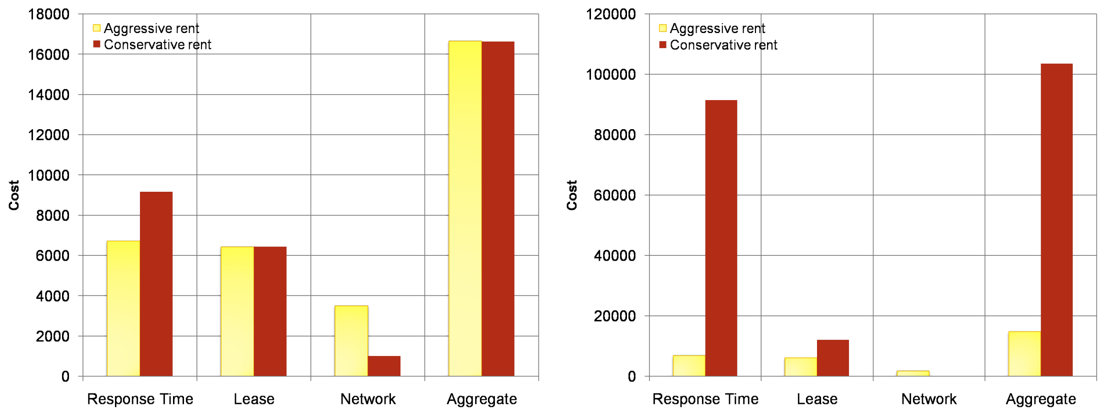


Figure 5.4: Cost breakdown for alternative scheduling schemes under VA system.

5.6.4 Impact of Conservative Approach

In previous experiments, both rent and release policies were based on an aggressive strategy in which the number of additional nodes to rent, or the number of rented nodes to release, is determined according to the number of queued jobs. In this section, we aim to examine an alternative rent policy that is based on a conservative strategy that only rents one node at any one time iteratively at each leasing decision. We aim to examine whether the conservative approach would be more beneficial to reduce the lease cost when compared to the aggressive policy, which we have proposed in the previous section.



(a) Cost breakdown for $\Omega = 1$.

(b) Cost breakdown for $\Omega = 10$.

Figure 5.5: Cost breakdown for aggressive rent and conservative rent policies.

The description of the conservative rent policy is presented in Algorithm 3, which is a straightforward modification of our previous algorithm. The result of applying the conservative rent policy is shown in Figure 5.5. It is interesting to note that the conservative rent policy incurs a slightly lower aggregate cost compared to the aggressive rent policy at rental threshold $\Omega = 1$. However, the aggregate cost difference is so negligible as to not provide any significant benefit over the aggressive policy. In fact, the aggressive rent policy gives a lower mean response time when compared to the conservative rent policy.

Algorithm 3 The conservative rent policy is slightly modified from that of the algorithm 2.

```
The implementation of the RENT function is as follows:
Let the $TP$ be total number of nodes requested by $J.size$.
Let the $RN$ be the number of nodes to rent.
If( $TP > F.size$ ) then
    1. Rent $RN=1$
    2. Waits for $RN$ nodes to arrive.
```

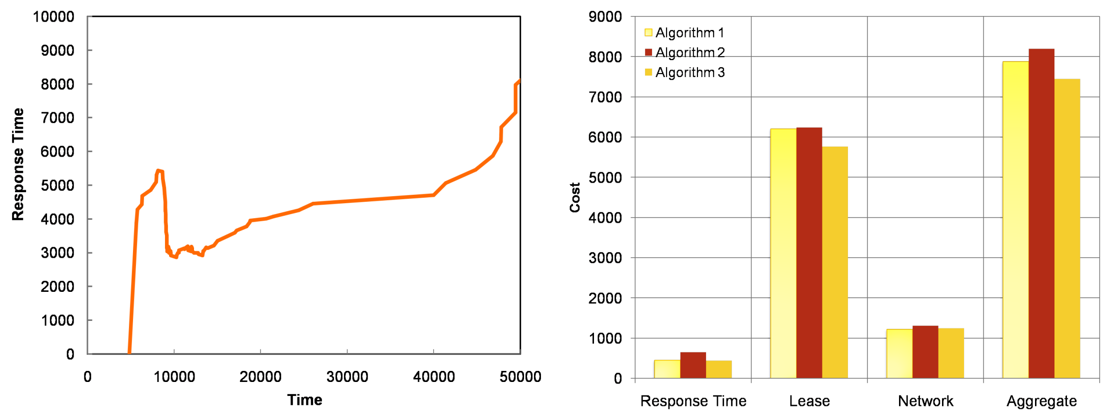
Quite interestingly, in contrast to the results in Figure 5.5a, Figure 5.5b shows that the aggressive policy is marginally better than the conservative policy for $\Omega = 10$. In particular, we observe that the mean response time for the conservative rent policy is 11 times worse than the aggressive rent policy. Moreover, the lease cost is 9 times lower for the aggressive rent policy when compared to the conservative rent policy. Again, the only cost bottleneck for the aggressive policy is the high network cost. When compared to the conservative rent policy, the network cost for the aggressive rent policy is 2.5 times higher than the conservative rent policy. Nevertheless, the aggregate cost for the aggressive rent policy is still significantly lower than that of the conservative rent policy by 7 times. This in itself is a compelling reason to recommend the aggressive rent policy over the conservative rent policy.

5.6.5 Impact of Release Policies

We have so far investigated the impact of rent policies under varying loads and threshold rental events. We have observed that the performance of such rent policies is heavily influenced by these rental threshold events. Our next research question is: *"What is the impact of releasing rented nodes (returning nodes to owners) at the end of job completion or release threshold events?"* Intuitively, we would expect some significant lease cost savings by relinquishing idle nodes, but the effect of releasing too many nodes may have a detrimental effect on cost due to (i) high response times when there are insufficient nodes to serve queued jobs and (ii) high administrative costs if nodes are rented and released in quick succession.

Algorithm 4 The selective release policy is also a slight modification of the original algorithm 2.

```
Response Time target: $\mho = 9984.93$
The handling of a task's completion is:
    1. Increase the $F.size$ capacity by 1,
    2. If( $rpt < \mho$ ) then
        (a) Invoke the RELEASE function.
```



(a) Response Time. (b) Cost breakdown.

Figure 5.6: The impact of release policy for $\Omega = 10$ and $\mho = 9984.93$.

In order to examine the impact of release policies, we again make a slight modification to our previous algorithm. Algorithm 4 shows a selective release policy that aims to optimize the lease cost within a specific target on the response time. For the purpose of the next experiment, we set the response time target to 9984.93 seconds while all other parameters are kept to their default values. The response time target of 9984.93 seconds is derived from an average response time of a dedicated HPC system (see Figure 5.2a). The aim is to show that the system is able to satisfy the mean response, within the response time target, i.e., 9984.93 seconds at a possibly cheaper cost.

Figure 5.6a shows that the response time is tuned iteratively to stay below 9984.93 seconds. Initially, we see that before the response time goes beyond 8000 seconds, it sharply decreases and then it steadily remains around a response time of 3000 seconds. Similarly, upon release decisions, the response time increases sharply until it gets to the approximate point of 5500 seconds. At later points, the response time steadily increases, but it never reaches above the response time target of 9984.93 seconds. The result indicates that the proposed release policy has successfully optimized the lease cost, in an iterative fashion, and it does so within the response time target of 9984.93 seconds and at a lower aggregate cost compared to our previous Algorithm 2. Figure 5.6b further shows a more detailed comparison of the response time, the network, and lease costs for Algorithms 2, 3 and 4, respectively, which further demonstrates the effectiveness of the proposed release policy.

5.7 Related Work

In this chapter, we pursue two main goals: (1) to analyze the total of ownership and the different types of costs involved in the construction and operation of current distributed resource infrastructures and (2) to use this analysis to examine the cost-effectiveness of such approaches under different distributed and high performance computing environments.

Despite the significant amount of research work on distributed and high performance computing, only a few works have been published on the costs caused by high performance computing and distributed computing infrastructures. In Deelman et al. (2008), the authors determine the cost of running a scientific workflow over a Cloud. They find that the computational costs outweigh storage costs for their Montage application. In Palankar et al. (2008), the authors consider the Amazon data storage service S3 for scientific data-intensive applications. They conclude that monetary costs are high as the storage service groups availability, durability, and access performance together. Furthermore, Greenberg et al. (2008) examined the costs of Cloud service data centres. The cost breakdown reveals the importance of optimizing work completed per dollar invested. Kondo et al. (2009) determined the cost-benefits of Cloud computing versus volunteer computing applications. The authors calculate the volunteer computing overheads for platform construction, application deployment, compute rates, and job completion times. They concluded that volunteer computing can provide an alternative solution to Cloud systems.

In Opitz, König, and Szamlewska (2008), the authors conducted a general cost-benefit analysis of Grid computing. The authors estimate the costs in two real-life Grids: the EGEE project and the Grid of the pharmaceutical company Novartis. However, no specific type of scientific application is considered. The costs only measure the total cost of ownership, but the application performance is not considered. Furthermore, no cost evaluation is made on a private resource system or a large dedicated HPC cluster.

In contrast, and for comparison, we consider different types of distributed computing infrastructures running from private systems to large dedicated HPC systems. To our knowledge, this work is one of the earliest attempts to examine the cost-benefit of small and large dedicated distributed computing systems over real-life Grid production systems. Furthermore, our work is the first to examine the cost effectiveness of a private resource system that offers the possibility for continuous usage of equipment, but with the ability to rent resources from real-life EGEE Grid production systems when more resources are needed to accommodate user demands.

If Grid resources are to be rented out, an efficient decision support is needed for the planning of resource usage. The works of Popovici and Wilkes (2005) and AuYoung et al. (2006) address the question of under which conditions additional resources should be acquired from resource providers. They address the area of resource commercialization in a service-oriented, utility-computing, and Grid-like world. They explicitly consider resource costs as part of their evaluation and they identify the necessity for rental decision support. The objective of such rental decision support is to maximize the profit of a service provider. Further, Burge, Ranganathan, and Wiener (2007) present algorithms in systems that make use of job revenues to maximize the service provider's profit against that of resource costs. In contrast, and for comparison with previous works, we carried out detailed cost analysis on a small-scale private resource system and on large-scale maintenance systems such as a large dedicated HPC system and a real-life production grid. We further carried out an evaluation that considered both the application performance and total cost of ownership of building distributed computing infrastructures, which, to the best of our knowledge are issues that have not been considered in previous work.

5.8 Chapter Summary

Our main focus in this work is on the evaluation of the cost effectiveness of renting Grid resources from a real-life production Grid system over a dedicated HPC system. A cost-benefit analysis of small and large HPC systems as well Grid computing was conducted in order to achieve this objective. We determined the cost-benefits of a dedicated HPC system versus a VA system. We detailed the specific costs of a private resource system, a dedicated HPC system and the EGEE Grid project. Our work also goes further as we make use of financial expense data from the EGEE Grid project and the workload traces from a dedicated LLNL HPC system to examine the performance and cost effectiveness of our VA approach, i.e., to what extent it improves cost and performance when compared to a dedicated HPC system.

From our analysis, we found that there is a potential cost-benefit in making use of Grid computing nodes for running computational jobs. With monetary cost-benefits in mind, we then presented a new VA approach for the effective use the Grid computing resources in running computational jobs. We then presented the results of our experiments, which were based on a simulation of real workload traces collected from a LLNL HPC system. Our experimental results indicated overwhelmingly that the VA system delivers substantial cost savings, while providing better performance in comparison to a dedicated HPC system. Our results, derived from an analysis of LLNL workload data, also provide an insight into the effectiveness of the VA approach:

- In a dedicated HPC system environment, the aggregate cost increases as the load increases due to high response times. However, the cost to operate a VA is not greatly affected by increased loads. This finding demonstrates the practicality of such a VA approach when there is both low and high demand for resources.

- From the examination of both aggressive and conservative rent and release policies, our results demonstrated that the aggressive rent policy consistently performs better than the conservative policy. The conservative rent policy performs worst at high rental threshold events. Furthermore, greater cost-benefits can be obtained with only minor modifications by applying the selective release policy.

We conclude that there is good evidence to show that the Grid resources can be effectively utilized for job execution by the VA system. We have demonstrated the effectiveness of a VA system in that it can operate at significantly lower costs and still provide high performance computing solutions for scientific users. This illustrates that a VA system can be a viable alternative for consumers that do not have the spending power for the construction and maintenance of a dedicated high performance computing system.

Chapter 6

Rental Policies

In the previous chapter, we demonstrated that there is a potential cost-benefit in adopting a small private resource system with the ability to rent processing power based on workload demand. This finding has led to the proposal of a Virtual Authority (VA) system that can provide new avenues for agility, service improvement and cost control. We have briefly examined two simple rental policies, namely, the aggressive policy and the conservative policy, that can be effectively used to guide rental decisions based on the number of queued jobs and the number of rented resource nodes in the VA. The experiments have shown that the provision of rental policies can have significant impact on the scheduling performance and resource cost.

Although the preliminary results are encouraging, the examined rental policies are static in nature: the threshold values used for rental decisions are fixed and they do not change dynamically in response to fluctuating resource demands. As a result, the rental policies may not be efficient under changing system conditions. To resolve this issue, this chapter presents extensions to the aggressive and conservative rental policies to incorporate dynamic information of system load and job deadline information when performing rental decisions. We then propose several cost-aware policies that take into account job monetary value and resource cost to further improve rental decisions.

Furthermore, our performance metrics in the last chapter do not take into account deadline violations from potentially deadline-driven jobs. As a result, the preliminary results do not fully capture the performance from the user's perspective. Given this limitation, this chapter first presents a costing model that takes into account the quality-of-service (QoS) as part of the performance metric. The costing model simplifies the performance evaluation because it only uses a single performance metric to capture the overall effectiveness of a rental policy based on its ability to increase application QoS requirements and to reduce resource cost.

6.1 Costing Model

In the previous chapter, our performance metrics consider the trade-off between response time and the cost to rent computational resources. However, it is very difficult to make cost-benefit comparisons of the different rental policies without having a single cost metric. The response time metric cannot be simply included in the administrative and rental costs since the response time does not have a direct relationship with the cost of resources. Therefore, we aim to extend our costing model to combine the performance aspect (i.e., response time) with the resource costs. We intend to capture both performance and resource costs factors with only a single cost metric.

In achieving this, the revised costing model makes use of an economics-based approach whereby applications express the monetary value of their jobs as the price they will pay to have them run, and the gap between this price and the cost to run the job is simply the job's 'profit'. The profit is used as the main evaluation metric, which is the trade-off between earning monetary values, penalty for deadline violation, and paying for the rental cost. The profit rigidly defines a single metric that measures the overall cost-effectiveness of the resource management system. Given this single metric, the goal of the VA is to increase its profit as much as possible.

We first assume the rental cost for resource *n* to be linear:

$$RL_n = \zeta_n + \eta_n(t_r - t_c) \tag{6.1}$$

where ζ_n is an administration fee and η_n is a consumption charge which is a fee paid per time increment. A rental fee is charged from the moment the scheduler takes control of the resource node (t_c) until the resource node is returned to the resource provider or its owner (t_r).

The administration fee ζ_n is a one-time overhead renting cost. It can represent the setup overhead cost to deploy, run or execute applications, operating systems or virtual machines. This may include the initial data that need to be loaded or transferred from the appropriate data sources. For instance, the overhead may come from the configuration and the deployment of new nodes from the resource providers, and/or the associated cost from the deployment of process virtual machines like PVM (Beguelin et al., 1991) or operating system-level virtual machines like XEN[1] (Barham et al., 2003b), and/or the associated cost from network bandwidth usage which can also potentially affect the system performance. Similarly, the administrative cost may also represent a cancellation charge for releasing a node. Therefore, the administrative cost ζ_n prevents the system from renting and releasing nodes in quick succession.

The consumption η_n unit cost is constant and it does not change during the duration of the rental agreement. However, the rental component η_n may well be influenced by 'utility', making popular resources (nodes) more or less expensive during the day. The time it takes for each node to arrive is either constant or is exponentially distributed.

We consider a resource sharing model whereby nodes are obtained from resource provider(s) without a time constraint[2]. However, due to the operating costs incurred from maintaining such nodes (e.g., due to electricity and system administrator' costs), it is to the benefit of the VA to release idled and unwanted nodes if they are no longer needed. Therefore, effective planning is somewhat required to ensure that the rental cost is not tied up unnecessarily. Following this, the revenue from the application

[1]There is no reason why a general operating system-level machines like KVM and XEN (see chapter 2) cannot co-exist with process virtual machines like PVM in this system. The only difference is that nodes or machines are abstracted as a service and given to customers. This is no different from taking machines out of service for maintenance or when machines are faulty. The owner loses usage of the machines and therefore is paid a rental fee. The customers on the other hand have the flexibility as to how they want to utilize the hardware.

[2]In Chapter 7, we will relax this assumption to consider a mutual agreement between the VA and the resource provider to provide a service (node) for a fixed rental duration at an agreed cost.

for request j is given as

$$RV_j = q_n + p_n(t_r - t_u) - \delta_j \tag{6.2}$$

where q_n is a fixed overhead charged for renting a node, p_n is a usage fee on node *n*; a fee paid per time increment from the moment request j is able to use the node (t_u) until the node is returned upon request completion (t_r). The penalty δ_j is incurred if the system fails to deliver a deadline contract for request *j* agreed with the customer application. Such a contract is an agreement for the VA to deliver the resource and complete request j before a given deadline D_j. The VA is penalized with the penalty cost for the failure of not meeting a given deadline. The penalty cost must be kept minimal to ensure that the maximum revenue can be obtained.

A resource *n* is defined to the management system by its rental-cost vector $[\zeta_n, \eta_n]$ and to the applications by the vector $[q_j, p_j, \delta_j]$. The system makes a loss if $\sum_{j=1}^{J} RV_j - \sum_{n=1}^{N} RL_n < 0$ where *J* is the total number of jobs submitted in the system and *N* is the total number of rented nodes.

Most related studies measure the performance of systems using metrics such as wait time, response times, utilization, and bounded slowdown, where the average is calculated over the jobs in the simulation. In Chapter 5, we use the mean response time to compare the performance and cost-benefits of a rental management system in contrast to those of a private resource system. Here, we revised our costing model slightly for the same purpose: to capture all of these characteristics with a single *profit* metric to simplify performance comparisons of the different renting policies. The advantage of the profit metric is that it represents a single value that captures overall trade-offs between satisfying the customer applications and the cost of renting external resources.

In summary, we compute the following for the performance metrics:

$$Revenue = \sum_{j=1}^{J} RV_j, \tag{6.3}$$

and

$$Rental\,cost = \sum_{n=1}^{N} RL_n, \tag{6.4}$$

and finally

$$profit = \sum_{j=1}^{J} RV_j - \sum_{n=1}^{N} RL_n \tag{6.5}$$

6.2 Workloads

Using a discrete-event simulator, we evaluate the proposed policies using real workload traces collected from the Parallel Workload Archive (D.G.Feitelson, 2009). In this section, we give details of the workload used to drive the simulator, the parameters that are varied in this exercise, and the descriptions on how the performance metrics of interest are captured by our discrete-event simulator. The workload traces are constructed as input for the event-driven simulation. The simulation responds to different types of events when updating the performance metrics. In this section, we describe the experimental setup we used, and the default parameters used in our experiments.

For each experiment, we choose a subset of the first 10,000 jobs out of 100,000 jobs from the original LLNL Thunder workload trace. A subset of 10,000 jobs from the LLNL trace is chosen from the period of Feb 2007 until June 2007. From our analysis, the workload contains an average of 42 processors, average inter arrival time of 30 seconds (0.5 minutes), and average runtime of 2,186 seconds (36.4 minutes). These LLNL workload traces were chosen because the workloads contain a large number of smaller to medium-sized jobs that are used by a large number of HPC users (283 users) with different and dynamic processing computing requirements (from large numbers of smaller to medium-sized jobs). The default parameter settings are shown in Table 6.1.

Parameter	**Default Value**
Number of Jobs	10,000
runs per data point	20
interarrival time	0.5
low:high job value ratio	20:80
urgent:non-urgent ratio	80:20
low:high job value factor	4
low:high job deadline factor	4
mean node arrival delay	1.0
penalty value	-monetary value
resource cost per node (per hour)	0.26

Table 6.1: Default simulator parameter settings.

To our knowledge, none of the real workloads available contains information pertaining to job urgency or the amount the application is willing to pay for it. We therefore follow a similar methodology to that in Irwin, Grit, and Chase (2004); Islam et al. (2007); Yeo and Buyya (2007) to synthetically model these parameters through two job classes: (i) high urgency and (ii) low urgency. Our workloads consist of *X%* urgent tasks and *100-X%* non-urgent tasks. Following Yeo and Buyya (2007), the *deadline factor* between urgent and non-urgent job is stretched by a default value factor of 4. In the later experiments, we evaluate the impact of cost under varying deadline factors.

Similarly, we also use the same methodology to model the monetary values. The monetary value for each job is categorized based on their urgencies. For an urgent job, a high monetary value is assigned using a normal distribution with mean 1.0 and standard deviation of 0.2 ($\frac{1}{5}$). On the other hand, a low monetary value is assigned using a normal distribution with mean 0.25 and standard deviation of 0.05 ($\frac{0.25}{5}$) for a non-urgent job.

The two values of the standard deviation parameter are chosen to ensure that the generated monetary values fall within the positive number, i.e., they do not become negative. Using this model, we are able to differentiate 'expensive' and 'cheap' jobs. For instance, in this case, the gap between expensive and cheap jobs is stretched by a *monetary factor* of 4. Such an approach realistically reflects real application scenarios whereby an application is more likely to pay a high value for an urgent job (e.g., strict deadline

Notation	Definition
S_j	amount/number of nodes requested for job *j* (job size)
φ_j^a	arrival time of job *j* in the system
φ_j^s	the time when job *j* is allocated with requested nodes
φ_j^p	processing unit for job *j*
φ_j^e	execution time for job *j*
φ_j^c	completion time for job *j*
φ_j^w	wait time for job *j*
φ_j^r	response time for job *j*
V_j	monetary value for job *j*
D_j	deadline for job *j*
δ_j	penalty cost for job *j*
η_n	consumption charge for node n
ζ_n	administrative charge for node n
ψ_n^a	arrival time of node *n*
ψ_n^t	termination time of node *n*
β_n	speed factor for resource node *n*
R	number of nodes *n* to rent
A	total amount of currently rented nodes of type *n*
F	amount/number of free nodes in the VA

Table 6.2: Main notations used for the simulation model.

job or short running job). Likewise, it is also more likely to pay a low value for non-urgent job (e.g., best-effort or long deadline job). Furthermore, each time a job fails to complete within its deadline D_j, the penalty value imposed is equal to the total value of the monetary job ($\delta_j = D_j$). We assume that the cost to rent a resource node from the resource provider is 0.26 per hour. This rental cost value is obtained from our cost-benefit analysis in Chapter 5. Finally, for statistical significance, each experiment is repeated 20 times. Therefore, the results shown in the experiments are the average values of measurements obtained in 20 runs. Appendix B describes our workload generator in more detail.

For each experiment, the VA Scheduler is responsible for job scheduling, whereas the VA Negotiator is responsible for rental decision making. The job requests arrive into the VA Scheduler and are served in a simple FCFS fashion, unless stated otherwise. We also assume that the scheduling overhead is negligible. This is a reasonable assumption, given the simplicity of algorithms that are considered. Moreover, the effects of the memory requirements and the communication latencies are not considered explicitly. Instead, they appear implicitly in the form of job execution time functions.

6.2.1 Simulation Model

In this subsection, we formally present our simulation model that captures the performance metrics of the VA Scheduler, VA Negotiator, and Resource Nodes. The simulation model consists of a set of rules that govern measurements between the VA Scheduler, the VA Negotiator and Resource Nodes components. The main notations used to describe the simulation model are shown in Table 6.2.

VA Scheduler. For each new incoming job request, we record the following: V_j: monetary value for job request j, S_j: number of nodes requested by job request j, arrival time φ_j^a for job request *j*, serve time φ_j^s for job request j, completion time φ_j^c for job request *j*, and F: free resource nodes currently available in the system. If number of requested nodes S_j is greater than F, the VA Scheduler triggers a rental event to evaluate the rental decision. Otherwise, the request is served with its requested number of nodes S_j.

Each time a job request is assigned to its number of requested R_j nodes, the wait time φ_j^w for request job j is computed ($\varphi_j^w = \varphi_j^s - \varphi_j^a$). Furthermore, each time a request has finished its execution, the job completion time φ_j^c is recorded and the job execution time is computed as $\varphi_j^e = \varphi_j^s - \varphi_j^a$. From the wait time and the execution time, the response time is calculated as $\varphi_j^r = \varphi_j^w + \varphi_j^e$. Alternatively, from the recorded completion time, the response time can also be calculated as $\varphi_j^r = \varphi_j^c - \varphi_j^a$. Finally, if a job request's response time is longer than than a request's deadline ($\varphi_j^r > D_j$), the penalty cost δ_j for request *j* is imposed.

VA Negotiator. Each time a VA Negotiator rents a node, we record the following: start rental time R_n for each node *n* and the amount (number) of nodes R_n to rent from the resource provider. Each time a node arrives into the system, we record the arrival time ψ_n^a for node *n* in the VA. Alternatively, each time a rental termination for node n has been issued, we record the termination time ψ_n^t for node *n* in the VA (the time when the node is released). Further, we record any additional charge ζ_n that is incurred due to fixed charges from overhead cost, administrative usage cost, and/or rental cancellation fee for the node *n*. Consequently, the total rental cost for the node *n* is $\zeta_n + \eta_n([\psi_n^t - \psi_n^a])$, which is computed each time the node *n* is returned to the resource provider.

Resource Node. A node consists of two parameters: processing speed and speed factor. A node can only be allocated to one job request at any one time. For the sake of brevity, it is assumed that each node consists of a single processor only. With such an assumption, the job execution time can be calculated from the node's processing speed. The speed factor β_n for resource node *n* can be used to assess the capacity of all resource nodes against a standard node by defining a standard node and a standard job. For example, consider a job *j* with processing units of 10 ($\varphi_j^p = 10$). If the speed factor β_n is set to two, the time it takes to complete job *j* is $\varphi_j^e = \varphi_j^p / \beta_n$ (i.e. $5 = 10/2$).

In summary, the total revenue from the applications for total *J* jobs submitted is:

$$RV = \sum_{j=1}^{J} [V_j . \varphi_j^e] - \delta_j \tag{6.6}$$

The total rental cost for all rented resource nodes *N* is:

$$RL = \sum_{n=1}^{N} \zeta_n + \eta_n([\psi_n^t - \psi_n^a]) \tag{6.7}$$

Finally, the profit or the net revenue is calculated as:

$$Profit = RV - RL \tag{6.8}$$

6.3 Baseline Performance

We begin with a set of exploratory runs that are designed to set out the behaviour of the system under relatively straightforward conditions, before proceeding to more challenging situations. This first set of experiments establishes a baseline operating environment under a VA system that dynamically rents resources on demand to meet the changing QoS requirements of applications.

In Chapter 5, we have already investigated several variants of rental policies based on aggressive and conservative strategies. Such policies keep track of the number of requested nodes for each job arrival in order to determine the number of nodes to be rented. However, these policies face a serious limitation since the threshold value has to be statically defined. Therefore, they may not be efficient when the load fluctuates at runtime. We therefore propose an extension to the previous aggressive and conservative policies to allow the threshold value to be changed dynamically in response to changing resource demand.

We now describe our extensions to these policies in detail. Each rental policy has a combination of *rent()* and *release()* operations. The *rent()* operation rents additional nodes, whereas the *release()* strategy releases nodes from the system. The *rent()* operation is invoked when a specific event has occurred or at periodic time intervals. Once the *rent()* operation is triggered, the system waits for nodes to arrive. The subsequent nodes are then rented by the system until the *release()* operation is invoked.

The rental threshold value changes dynamically based on the estimation of a single parameter at time T as the average of the last m observations, where m is the weighted moving average interval and it decreases by 1 with each previous observation. Further, the value of m represents the average number of requested nodes (job size) from each queued (waiting) job. The weighted moving average will provide an estimate of the mean of the observations if the mean is constant or slowly changing. In the case of a constant mean, the largest value of m will give the best estimates of the underlying mean. A longer observation period will average out the effects of noise and variability. The idea is to use the average requested number of nodes by all queued jobs as the representative guideline to determine the appropriate rental threshold in order to prevent the system from over-reacting to transient load fluctuations.

The aggressive and conservative schemes can be incorporated with the *rent()* and *release()* operations to form four different rental policies: *aggressive rent()+aggressive release()*, *aggressive rent()+conservative release()*, *conservative rent()+aggressive release()*, and *conservative rent()+conservative release()*. The *aggressive rent()* scheme rents nodes based on the number of nodes requested by jobs in the pending or waiting queue (queued jobs). The aim is to serve more jobs in the queue but this could come at the expense of rental cost. Conversely, the *conservative rent()* scheme only performs renting when the total amount of rented nodes reaches a certain low level. The scheme aims to reduce the amount of nodes rented in order to reduce the rental cost but this could come at the expense of the penalty cost if there are insufficient nodes to handle newly arriving jobs.

Figure 6.1, 6.2 and 6.3 show the pseudo-code for the aggressive and conservative policies. Initially, the *aggressive rent()* or the *conservative rent()* operation is invoked at the end of job scheduling (line 11 at **SCHEDULE()** operation). For *aggressive rent()*, the weighted moving average is computed. Then, each waiting job is checked in the queue to determine the total number of requested nodes for all queued jobs (line 2-4 at **AGGRESSIVERENT()** operation). Subsequently, the total number of requested nodes is

Variables

- K : a set of information pertaining to the total number of requested nodes for all queued jobs.
- A : the total amount of currently rented nodes.
- F : the amount of free (available) nodes ready to be used.
- S_j : job size or the amount of nodes requested for job *j*.
- R : the amount of nodes to rent.

Functions

- **getLastMA()**: obtain the previously calculated moving average value. Returns null if there is no calculated moving average value is found.
- **computeMA(param)**: calculate the weighted moving average from a series of *param* values.

Invoked when a new job arrives.
ReceiveJob()

```
1: Insert new arriving job into the FCFS job queue
2: Determine the total number of requested nodes for all queued jobs ($\sum S_j$) and includes in K
3: Calculate the new moving average value based on past jobs and the new job arrival.
4: Invoke the SCHEDULE() function.
```

Invoked periodically or when a new job arrives.
SCHEDULE()

```
1: if JQ is empty then
2:     return;
3: end if
4: for $j \leftarrow 0$ to JQ do
5:     if $HQ.size \geq S_j$ then
6:         $F \leftarrow F - S_j$;
7:         remove j from JQ;
8:         Start executing j;
9:     end if
10: end for
11: invoke aggressive rent() or conservative rent();
```

Figure 6.1: Pseudo-code of the Aggressive and Conservative policies - Part 1.

```
Invoked when a task of a job has completed its execution.
TASKCOMPLETION()
1: $F_n \leftarrow F_n + 1$

Invoked when a job (all its tasks) has completed its (their) execution.
JOBCOMPLETION()
1: Update cost statistic and performance

Determine the condition when to rent and the number of nodes to rent.
AGGRESSIVERENT()
1: $MA$ =computeMA($V_j$);
2: for $j \leftarrow 0$ to JQ do
3:    $TS \leftarrow TS + S_j$;
4: end for
5: if $MA > TS$ then
6:    $R \leftarrow MA - TS$;
7:    if $R > 0$ then
8:       $hasOrdered \leftarrow true$;
9:       start to rent $R$ amount of nodes;
10:      waits for R nodes to arrive;
11:   end if
12: end if

Determine the condition when to rent and the number of nodes to rent.
CONSERVATIVERENT()
1: for $j \leftarrow 0$ to JQ do
2:    if $HQ.size < S_j$ then
3:       $TS \leftarrow TS + S_j$;
4:    end if
5: end for
6: if $TS > 0$ then
7:    $R \leftarrow TS - F$;
8:    start to rent $R$ amount of nodes;
9:    waits for R nodes to arrive;
10: else if $A < TS$ then
11:   start to rent $TS$ amount of nodes;
12:   waits for TS nodes to arrive;
13: end if
```

Figure 6.2: Pseudo-code of the Aggressive and Conservative policies - Part 2.

```
Invoked at the end of job completion and determine the number of nodes to release.
AGGRESSIVERELEASE()
1: for $j \leftarrow 0$ to $JQ$ do
2:     $TS \leftarrow TS + S_j$;
3: end for
4: if $A > TS$ then
5:     $R \leftarrow A - TS$;
6:     start to release $R$ amount of nodes;
7:     update $A \leftarrow A - R$;
8:     update $F \leftarrow F - R$;
9: end if

Invoked at the end of job completion and determine the number of nodes to release.
CONSERVATIVERELEASE()
1: $MA_{prev} \leftarrow \mathbf{getLastMA}()$
2: if $A > MA_{prev}$ then
3:     $R \leftarrow A - MA_{prev}$
4:     start to release $R$ amount of nodes;
5:     update $A \leftarrow A - R$;
6:     update $F \leftarrow F - R$;
7: end if
```

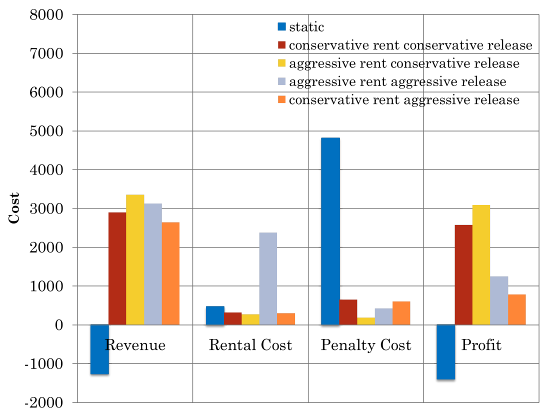
Figure 6.3: Pseudo-code of the Aggressive and Conservative policies - Part 3.

then checked against the computed weighted moving average value (line 5). If the moving average value is greater than the total number of requested nodes, the system initiates to rent additional nodes based on the moving average value and the total number of requested nodes (lines 5-11).

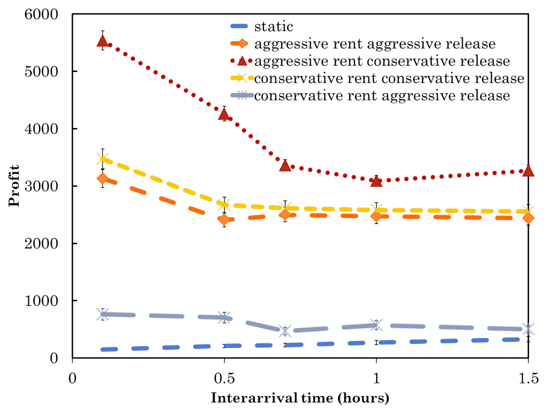
Unlike the *aggressive rent()* operation, the *conservative rent()* operation is only invoked where there are insufficient nodes to serve a job (lines 1-4 at **CONSERVATIVERENT()** operation). The number of free nodes is compared against the number of requested nodes for each job. Based on the additional number of requested nodes, the policy initiates the system to rent the resource nodes (lines 6-9). Furthermore, the policy also initiates renting of additional nodes if the current total number of rented nodes is fewer than the total number of requested nodes from all queued jobs (lines 10-12).

The *aggressive release()* scheme relinquishes all excess nodes based on the total number of currently rented nodes and the total number of nodes requested by all jobs (lines 4-9 at **AGGRESSIVERELEASE()** operation). The scheme aims to release all excess nodes in order to reduce the rental cost. On the other hand, the *conservative release()* scheme relinquishes nodes based on the last computed weighted moving average value. If the total number of rented nodes is higher than the last computed weighted moving average, the policy initiates renting of additional nodes (lines 2-7 at **CONSERVATIVERELEASE()** operation). The aim is to reserve slightly more nodes in the system than an average resource demand in order to allow more future jobs (especially urgent jobs) to be served.

We make an important assumption that the system may hold the resource node once rented for as long as desired at the expense of operating cost. Therefore, a *release()* call is required to release a node when it is no longer needed. The deployment cost prevents the system from rapidly renting and then releasing a node in quick succession. Furthermore, a simple FCFS scheduling scheme is employed to schedule job requests in the order of their arrival, unless stated otherwise.



(a) The cost breakdown for the static approach versus dynamic VA approach.

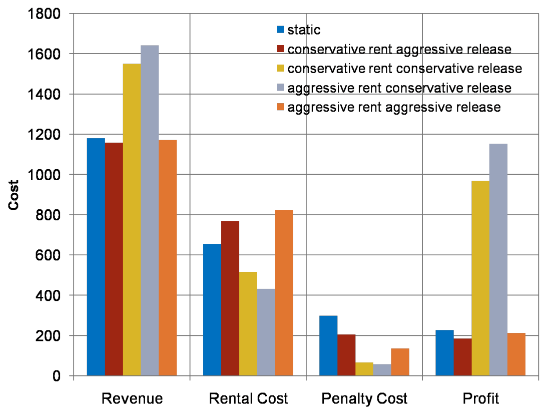


(b) Comparison of profits among the static and the rental policies under increasing interarrival times.

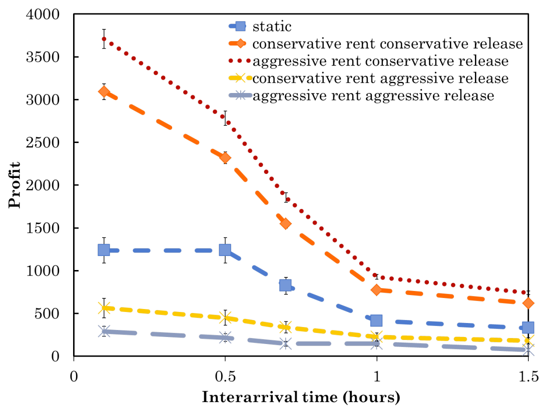
Figure 6.4: Comparison of rental policies under workload containing high percentage of urgent requests.

Figure 6.4a shows the comparison of the four different combinations of aggressive and conservative rental policies under a high ratio of urgent jobs for the following metrics: revenue, rental cost, penalty cost, and profit. The static policy approach (i.e., fixed resource pool) is also shown to set a baseline for comparison of the proposed rental policies under a dynamic VA approach. As can be observed, the *aggressive rent+conservative* release earns the highest profit, while the *conservative rent+conservative release* earns the second highest profit. The static policy earns the worst profit, while the *conservative rent+aggressive release* incurs the second worst profit, which is followed by *aggressive rent+ aggressive release*.

Figure 6.4b further shows the profits for the different rental policies as a function of increasing interarrival times (decreasing load). Our observation is that while the profit for the static system drops as the interarrival decreases (load increases), the profits for all four rental policies display an opposite trend. Interestingly, we can observe that the profits for all four rental policies increase as the interarrival time decreases (system load increases). This is attributed to the dynamic VA approach which offers the ability to rent and release resources at runtime. In a static policy where the number of nodes is fixed, a decrease in interarrival time (an increase in load) would result in lower profit due to higher penalty cost. Similarly, we can see that the *aggressive rent+conservative release* policy still earns the highest profit, while the *conservative rent+conservative release* policy earns the second highest. Also, as expected, the static system has the worst profit under all interarrival conditions, while the *conservative rent+aggressive release* and *aggressive rent+ aggressive release* policies have the second and third worst profits, respectively.



(a) Comparison of rental policies for workloads with lower percentage of urgent jobs



(b) Comparison of rental policies as a function of the interarrival time for workloads with relatively low percentage of urgent jobs.

Figure 6.5: Comparison of rental policies under workload containing low percentage of urgent requests.

To cross-validate our results, Figure 6.5a shows the cost breakdown and the profits for the different rental policies under workloads with a small percentage (20%) of urgent requests. Surprisingly, we observe a different performance trend that is not consistent with our earlier observation from previous experiments that were performed under a relatively high urgent requests workload. Interestingly, under workloads with a small ratio of urgent requests, we can see that the static policy no longer has the worst profit: the static policy earns better profit than both *aggressive rent+aggressive release* and *conservative rent+aggressive* release policies. The reason for the worst performance for the two policies may be attributed to the excessive release operations: the system is simply forced to rent additional nodes because it releases too many nodes in quick succession resulting in high administrative and rental costs. Renting and releasing too many nodes has the effect of high administrative cost (due to high set-up and cancellation costs) and we observe that the system finds itself renting more nodes on a frequent basis due to the effect from the aggressive release operation.

The results show that the *aggressive release* operation has quite a significant impact on the system performance under a lower percentage of urgent requests. On the other hand, we see that the performance of *aggressive rent+conservative release* and *conservative rent+conservative release* policies remains unchanged. We can still observe that the *aggressive rent+conservative release* policy earns the highest profit, while the *conservative rent+conservative release* policy earns the second highest profit.

Figure 6.5b further shows the results as a function of interarrival time. We can observe that all the four policies generally incur high profits as the interarrival decreases (load increases). From these results, our main conclusion is that both *aggressive rent+aggressive release* and *conservative rent+aggressive release* policies are not greatly affected by the decrease in the interarrival times (increasing load) but are more susceptible to variability in the percentage of urgent and non-urgent requests.

6.3.1 Incorporating Load

A high percentage of urgent requests is more likely to increase system load. Therefore, it is envisaged that profits can be improved further if the system load is taken into account when making rental decisions. Rather than just blindly renting nodes for each queued job when a rental threshold has occurred, we could restrict the system to rent extra nodes only when the system has a sufficiently high utilization. Similarly, nodes should only be released when system utilization goes below a certain threshold.

However, the system utilization cannot be evaluated too frequently so as to avoid overreaction to minor system and workload changes. Therefore, rental and release conditions should only be triggered when the utilization rate varies greatly[3]. Hence, we propose a policy that takes into account newly arriving jobs and issues rent operations when a specific utilization threshold has been reached.

The policy is described as follows: when the *rent()* operation is triggered, the system checks whether the current utilization has gone above the predefined threshold. For example, if the current system utilization exceeds the overdemand β utilization threshold value, additional nodes should be rented in accordance with the aggressive or conservative *rent()* scheme. Similarly, each time the *release()* operation is triggered, the system checks whether the current system utilization has gone below the predefined underdemand α threshold. Consequently, nodes can be released based on either the aggressive or conservative *release()* scheme. The utilization is defined as

$$util = \frac{\sum_{j \epsilon JQ} S_j . \varphi_j^e}{\sum_{n=1}^{N} RL_n}$$

where the numerator represents the total execution time for running all jobs on their allocated nodes and the denominator represents the total rental cost for all rented nodes.

Figure 6.6 shows the results for all rental policies at increasing urgent requests for $\beta = 0.7$ and $\alpha = 0.3$. Under the lower percentage of urgent requests condition, we can observe that all rental policies display comparable profits apart from the static policy that earns the lowest profit.

Interestingly, we also observe that the *aggressive rent()+conservative release()* policy no longer gains the highest profit. Instead, the *conservative rent()+conservative release()* policy earns the high-

[3]The reason is to provide quick response to workload changes while not overreacting to utilisation noise.

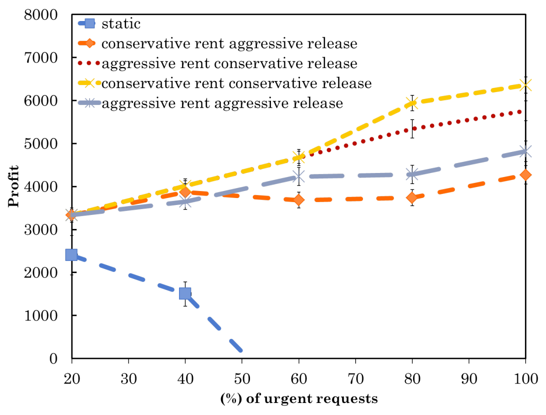


Figure 6.6: Comparison of static and all rental policies for $\beta = 0.7$ and $\alpha = 0.3$ under increasing percentage of urgent requests.

est profit, which is slightly higher than the *aggressive rent()+conservative release()* policy. A similar performance trend as that from the previous result is observed for both the *aggressive rent+aggressive release* and *conservative rent+aggressive release* policies: they both have better profits compared to the static policy.

From these observations, we conclude that the performance trend is similar except for the *conservative rent()+conservative release()* policy which has significant performance gains over other policies when the system load is incorporated as part of the rental decision-making process.

6.3.2 Incorporating Job Deadline

The previous policy accounts for the utilization threshold to determine when to rent additional nodes and when to release nodes. However, it does not consider the importance of individual job execution because job requests are not prioritized by their deadlines. We therefore attempt to enhance the performance by considering a policy that takes into account individual job deadlines.

Instead of employing the utilization threshold alone, the job deadline information can be incorporated into the policy system to reduce penalty cost, which in turn can improve the system profit. Given any two jobs, a job with a long deadline denotes more flexibility with respect to delays than a job with strict deadline. Therefore, non-urgent jobs with long deadlines can tolerate longer waiting time at the expense of more urgent jobs with strict deadlines. The aim is therefore to give greater priority to jobs with critical deadlines.

Algorithm 5 shows the pseudo-code for the rental mechanism that incorporates job deadline to determine the number of additional nodes to rent. The negotiator reacts when each job arrives with a corresponding completion time deadline requirement. When each job arrives, it attempts to schedule the new job. If the job cannot be served due to insufficient nodes, the negotiator determines whether there will be a violation of deadline if the job is going to be delayed in the queue (lines 3-5). If the job cannot tolerate further delay due to deadline violation, this job is marked as an urgent job and is pushed to the head of the queue (lines 5-9). This altering of job order within the queue is allowed until the violation threshold has been reached. When the violation threshold has been reached, the negotiator rents additional nodes as required for the job (lines 10-13). Similarly, if the job cannot tolerate further delay without violating its deadline, the job is immediately reserved while the system rents additional number of nodes to serve the reserved job.

Algorithm 5 The pseudocode for incorporating job deadline.

```
1:  for $j \leftarrow 0$ to $JQ$ do
2:      Set Penalty $\Gamma_j = 0$;
3:      if $\varphi_j^e + \varphi_j^w > D_j$ then
4:          $R \leftarrow S_j$;
5:      end if
6:      if $\Gamma_j > \Omega$ then
7:          $\Gamma_j = \Gamma_j + 1$;
8:          $R \leftarrow S_j$;
9:      end if
10:     if $R > 0$ or $\Gamma_j > 0$ then
11:         start to rent R amount of nodes;
12:         reserve job j with the allocated R amount of nodes;
13:     end if
14: end for
```

Figure 6.7 shows the performance for the different strategies of rental policies with increasing percentage of urgent requests. From Figure 6.7a, we can see a similar trend to that in the previous results with some additional observations. We can make the observation that at a low percentage of urgent requests, the profit generally increases when the deadline threshold increases. This is because the system is able to serve the jobs within their deadlines by renting a few nodes only, thus resulting in lower rental cost.

At a high percentage of urgent requests, we can see that the profit starts to fall significantly when the deadline threshold increases (Figure 6.7b). The result is attributed to the increase in rental cost. As can be observed, as the deadline threshold decreases, the rental cost increases.

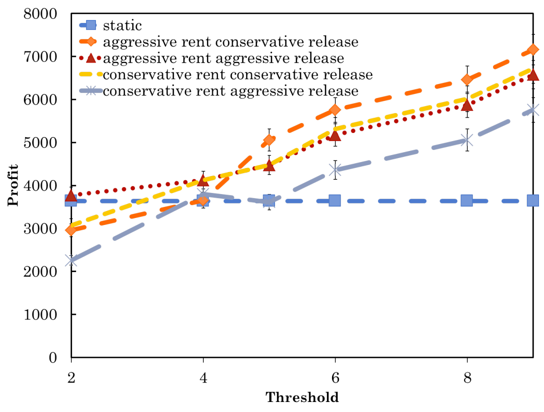
We can conclude that at a low percentage of urgent requests, the impact of increasing deadline threshold generates high profits. However, at a higher percentage of urgent requests, we observe an opposite trend: the profit falls significantly as the deadline threshold increases. This is attributed to the high penalty cost incurred. As the deadline threshold increases, the probability of the system renting sufficient nodes to serve queued jobs become very low. As a consequence, the penalty costs increase significantly which results in lower profits.

6.3.3 Summary

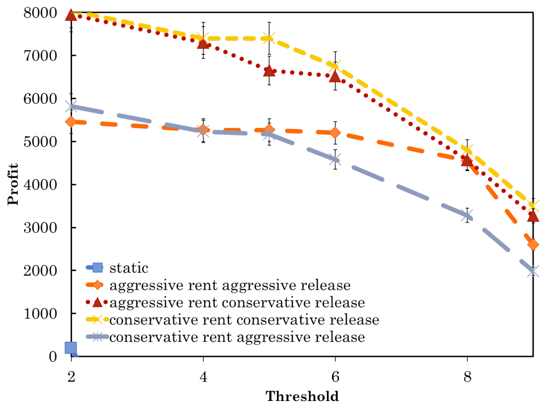
From the experimental results we have obtained so far, we can make the following observations:

- The profit for the static approach is always the worst and from the poor performance of both the *aggressive rent+aggressive release* and the *conservative rent+aggressive release* policies, we are able to conclude that the aggressive release scheme is not suitable for all workload conditions. On the other hand, the conservative release policy is highly recommended for workloads having either a small or large percentage of urgent requests.

- The *aggressive rent+conservative* release policy generally performs better than the other rent and



(a) 20% urgent requests.



(b) 80% urgent requests.

Figure 6.7: Impact of deadline threshold. The profits increase as the threshold increases for workloads having a small percentage of urgent requests. However, for workloads having a large percentage of urgent requests, high deadline threshold incurs lower profits.

release policies because it optimistically rents more nodes to serve queued jobs and at the same time it does not incur high rental cost.

- Although the *aggressive rent+conservative release* policy generally performs better, the *conservative rent+conservative release* policy generates higher profit for workloads having a higher percentage of urgent requests.

- Using both system load and job deadline information to guide rental decisions can significantly improve the profits, but this could also potentially lead to substantially lower profit if the deadline threshold is set too high under a high percentage of urgent jobs.

6.4 Cost-Aware Rental Policies

The aggressive and conservative rental policies investigated in the previous section do not make use of job monetary value or resource cost information when making a scheduling or rental decision.

It is envisaged that for a sustainable resource management system infrastructure, an economic-based or a cost-based scheduling framework reflecting operational costs, even using a virtual currency, is worth considering. Therefore, we present cost-aware rental policies that employ an economic-based approach when determining what jobs to schedule and when to rent resources. With cost-aware resource management systems, resource allocation is carried out based on an individual job's monetary value as opposed to conventional scheduling policies that optimize for system-centric performance metrics. Such policies may also take into account resource cost when performing rental decisions. We aim to examine the benefits of incorporating cost information when performing scheduling and rental decisions.

We first introduce the ValueFirst policy that makes use of simple job monetary value information when making rental decisions. We then introduce ProfitFirst and ConservativeProfit policies, respectively. Both ProfitFirst and ConservativeProfit policies take into account job monetary value and resource cost information to arrive at a renting decision. We compare them with the Greedy and DeadlineFirst policies which are described earlier in Section 6.3.1 and 6.3.2.

The Greedy policy is based on the aggressive rent and conservative release policies described in Section 6.3.1. Recall that with the Greedy policy, the system rents sufficient nodes for each queued job as soon as a rental threshold occurs. The aim is to fulfil as many job requests as possible. However, the Greedy policy does not make use of application QoS parameters such as job deadline and job monetary value information, which is certainly a disadvantage.

The DeadlineFirst policy also represents the rental policy previously investigated in Section 6.3.2. Such a policy considers the job deadline information to determine whether it is necessary to rent additional nodes. Upon a new job arrival, the system will rent additional nodes for queued jobs when a rental violation threshold has occurred, regardless of the rental cost. Both Greedy and DeadlineFirst policies will serve as the baseline for comparison with the other proposed cost-aware policies which we describe in more detail below.

6.4.1 ValueFirst

The ValueFirst policy aims to maximize the system revenue by renting nodes for all high-value requests. The policy classifies the job requests into two categories: urgent and non-urgent. A job that has the highest value is considered the most urgent request. To differentiate between high- and low-priority jobs, the negotiator differentiates job priorities in accordance with the job monetary value. A job with high value is considered an urgent job, whereas job with low value is regarded as a non-urgent job. To differentiate between low- and high-value jobs, the scheduler first determines the threshold value which is calculated based on an average monetary value from all submitted jobs. Upon each new job arrival, the scheduler determines whether the job value exceeds the calculated threshold value. If so, additional nodes are rented for the job. Otherwise, if the new job value is below the calculated threshold value, the job is marked as non-urgent and it is placed in the queue until a new job arrives. Algorithm 6 shows a sample implementation of the ValueFirst policy.

Algorithm 6 Implementation of the **ValueFirst** policy.

```
1: $MA$ =computeMA($V_j$);
2: for $j \leftarrow 0$ to $JQ$ do
3:   if $V_j > MA$ then
4:     $R \leftarrow S_j - F$;
5:     start to rent R amount of nodes;
6:     reserve job j with the allocated amount of R nodes;
7:   end if
8: end for
```

6.4.2 ProfitFirst

In all the proposed policies, the resource cost to rent nodes for executing the jobs is ignored. We now consider ProfitFirst policy that takes into account the cost of executing a request against the cost of renting additional nodes to serve that specific request. The basic idea is to rent additional nodes only when the total revenue can outweigh the cost of renting additional resource nodes to serve queued jobs. For each queued job, the system calculates the estimated total revenue based on the job size and the estimated job runtime. Using the estimated total revenue as our base guideline, we can estimate the least amount of resource nodes that can be rented without incurring excessive resource cost. Algorithm 7 shows a sample implementation of the ProfitFirst policy.

Algorithm 7 Implementation of the **ProfitFirst** policy.

```
1: for $j \leftarrow 0$ to $JQ$ do
2:   if $RV_j > RL_n$ then
3:     $R \leftarrow S_j - F$;
4:     start to rent R amount of nodes;
5:     reserve job j with the allocated amount of R nodes;
6:   end if
7: end for
```

6.4.3 ConservativeProfit

The ConservativeFirst policy is a variant of the ProfitPolicy that rents nodes as long the total resource cost does not exceed the total revenue. Similar to the ProfitFirst policy, it evaluates the cost of renting

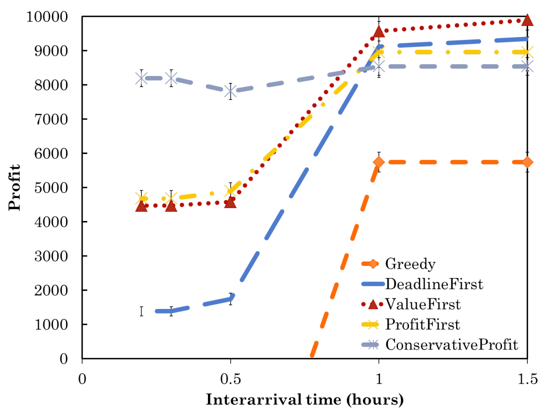


Figure 6.8: Impact of load.

extra nodes for each queued job request. However, unlike ProfitFirst, the ConservativeProfit also takes account of the total *net* profit when determining whether or not to rent additional nodes. Initially, all queued jobs are sorted in descending order according to their monetary values. For each sorted queued job, the system determines whether the total value from the system net profit including the estimated resource cost is lower than the estimated total revenue from the execution of the queued job. If the job passes this test, the system proceeds by renting additional nodes for that specific job. Algorithm 8 shows a sample implementation of the ConservativeProfit policy.

Algorithm 8 Implementation of the **ConservativeProfit** scheme.

```
1:  sort queue in descending order of monetary values.
2:  for $j \leftarrow 0$ to $JQ$ do
3:      Compute $RV_j$;
4:      Compute $RL_n$;
5:      if $RV_j > RL_n$ && $profit - RL_n \leq 0$ then
6:          $R \leftarrow S_j - F$;
7:          start to rent $R$ amount of nodes;
8:          reserve job $j$ with the allocated amount of $R$ nodes;
9:      end if
10: end for
```

6.4.4 Comparison of Cost-aware Rental Policies

In Section 6.3, we established our baseline results and demonstrate the feasibility of the aggressive and conservative rental policies. In the next experiments, we further examine the proposed five rental policies described in 6.4 that make use of cost information when making rental decisions. We evaluate the three cost-aware rental policies and compare them against the baseline performance of the Greedy and the DeadlineFirst policies. We evaluate all five rental schemes under different system configurations and workload conditions. Based on our previous findings that recommend the conservative release scheme, the *conservative release* operation is used for the subsequent experiments.

6.4.4.1 Impact of Load

Figure 6.8 shows the results for the different rental policies as the interarrival (load) changes. As the interarrival decreases, the system load also increases.

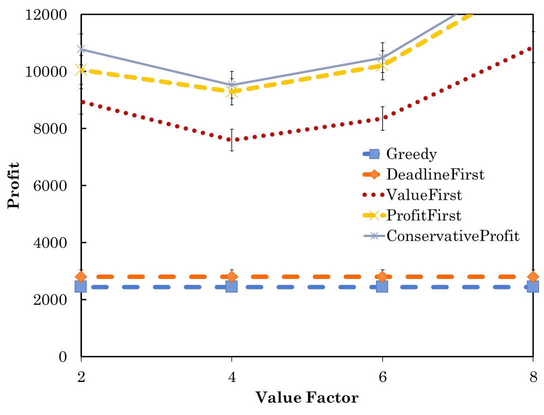


Figure 6.9: The effect of job value factor

For cost-aware policies, the effect of increasing the load results in an increasing profit as the system raises additional revenue from the increased load. We have also made several observations as follows: when the load is relatively low (interarrival ≥ 0.5), both DeadlineFirst and ValueFirst are able to achieve slightly higher profit than ProfitFirst and ConservativeProfit. This shows that higher profit can be achieved when both deadline and monetary value are taken into account when the load is relatively low. This is due to the fact that DeadlineFirst and ValueFirst exhibit lower penalty costs compared to ConservativeProfit and ProfitFirst under light load conditions. Furthermore, this also can be attributed to the nature of both DeadlineFirst and ValueFirst that aggressively rent additional nodes to satisfy the deadlines of urgent jobs. However, as the offered load increases, the profits for DeadlineFirst and ValueFirst drop significantly. It can be observed that, at high load (interarrival≤ 0.5), both ConservativeProfit and ProfitFirst start to aggressively outperform DeadlineFirst and ValueFirst. In particular, we can see that the profit for DeadlineFirst falls substantially. The trend continues as the interarrival time decreases (load increases).

6.4.4.2 Impact of Monetary Values

We examine the sensitivity of improvements in profit to variations in the job monetary values. The variation in monetary values is examined because the monetary values have direct impact on the system's revenue and its profit. If there were no variation in value, the cost-aware policies may not show any difference in profits since all jobs would be equally important and consequently providing differential resource allocations would not matter. We investigate the impact on profits when varying monetary values. For the experiment, we vary the job value mean factor from 1 to 8 while all other parameters are kept constant. The mean interarrival time of the jobs is also kept fixed to 0.5 hours.

Figure 6.9 presents the results. We observe that the cost-aware policies (ConservativeProfit, ProfitFirst, and ValueFirst) outperform Greedy and DeadlineFirst by a wide margin as the mean value factor increases. This confirms our hypothesis that higher gaps in value variations can lead to a significant increase in profits. However, the profits for Greedy and DeadlineFirst remain constant at varying monetary values – they are are not affected at all by the increase of mean value factor. This is due to the fact that they do not take into account the job monetary values when performing rental decisions.

6.4.4.3 Impact of Job Deadline

We examine profit sensitivity to variations in the job deadline factor. We vary the deadline factor from 1 to 16 and the job mean value is set to 10 while all other parameters are kept constant. As the deadline factor increases, the job becomes less urgent. The mean interarrival time of the jobs is also kept fixed to 0.5 hours.

Figure 6.10 shows the system profit as a function of the job deadline factor. The job deadline factor represents the increase in the length of the job deadline. We can observe that the increase in the job deadline factor does not lead to a significant improvement in system profit. Only a slight increase in profit is achieved. Nevertheless, we can observe that even when jobs are moderately insensitive to deadline, the cost-aware rental policies still deliver higher profits compared to the non-cost-aware (i.e., Greedy vs. DeadlineFirst).

We observe that ConservativeProfit and ProfitFirst still deliver significantly higher profits compared to Greedy and DeadlineFirst. For example, even under a strict deadline, we can still observe that ConservativeProfit and ProfitFirst cost-aware policies earn approximately a 60% increase in profits compared to the non-cost-aware DeadlineFirst policy. On the other hand, the ValueFirst policy earns a slight increase of approximately 30% increase when compared to the DeadlineFirst. As expected, the Greedy policy incurs the worst profit.

The difference in profits between ProfitFirst and ConservativeProfit can be directly attributed to the deadline and monetary value information that the cost-aware policies are able to use to guide renting decisions whenever resource competition occurs. It can be observed that the ConservativeProfit policy earns slightly higher profit than the other cost-aware policies, ProfitFirst and ValueFirst, because it is able to rent the right amount of nodes to serve queued jobs within their deadline constraints without incurring unnecessary rental cost.

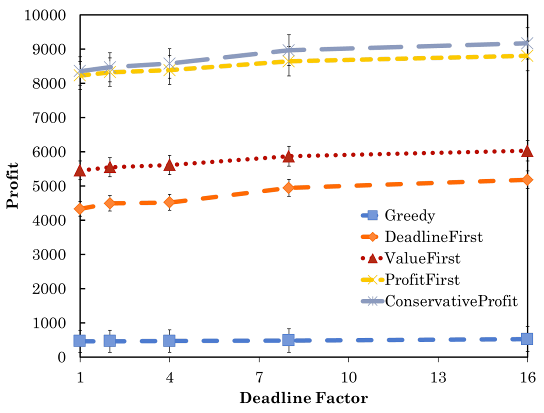


Figure 6.10: Impact of job deadline factor.

6.4.4.4 Impact of Node Heterogeneity

We evaluate the effect of the variation of the node capacity on the rental performance. A node consists of two parameters: processing speed and speed factor. A node can only be used by one job request at any one time. We assume that each node consists of a single processor only. The processing time or duration

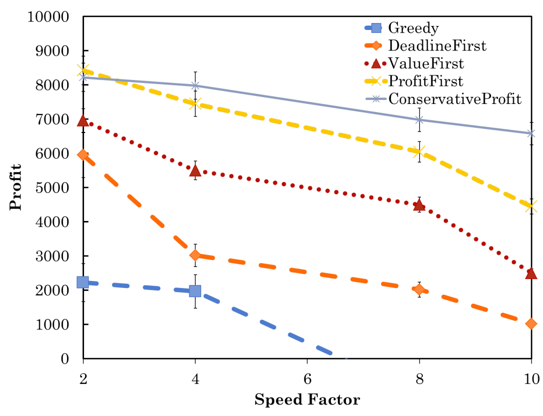


Figure 6.11: Impact of node heterogeneity.

would depend on the node's processing speed. Therefore, the speed factor β that represents the mean service rate of each processor can be used to assess the capacity of all nodes against a standard node by defining a standard node and a standard request.

We vary a node capacity between 2 and a speed factor of β. For example, consider a job request j with processing units of 10 (e.g., $S_j = 10$). If the mean speed factor β is set to 1, the time it takes to complete request j is approximately $\tau_j = S_j/\beta_N$ (i.e., $10 = \frac{10}{1}$).

Figure 6.11 illustrates the cost breakdown for the five examined rental policies under different variations of processor capacity. As observed, when the node capacities increase, we can see that the profit deteriorates for all policies. In all cases, the profits for ConservativeProfit and ProfitFirst are comparable under low speed factor. The ConservativeProfit generally obtains higher profit than the ProfitFirst and, as the speed factor increases, the gap in the profits between ConservativeProfit and ProfitFirst also widens. This is due to the effectiveness of ConservativeProfit in reducing rental cost while minimizing penalty cost. We can see that this is more evident in case of high capacity nodes (from $\beta = 8$ to $\beta = 10$). As expected, the Greedy policy earns the lowest profit, while the DeadlineFirst policy incurs the second highest profit.

6.4.4.5 Impact of Node Arrival Delay

We examine the impact of the node arrival delay or the node arrival time on our rental policies. The node arrival delay is the average delay (in minutes) that the system must tolerate before the node is ready once the rent operation is initiated. It is the time it takes for a rented node to be available for job scheduling and resource allocation upon rental request (i.e., duration of the period between rental request and rental completion).

Figure 6.12 shows the effect of increasing node arrival delay. The first observation is that a long arrival delay has severe effects on all rental policies; we can see that the profits for all policies diminish significantly as the node arrival delay increases. The ConservativeProfit and ProfitFirst earn the highest profits among all other policies when the range of node arrival delay is low (i.e., between 1 and 14 minutes). Interestingly, when there is a longer delay (above 16 minutes), we can see that the ValueFirst rental policy starts to outperform the ConservativeProfit and ProfitFirst policies. At this point, the profits

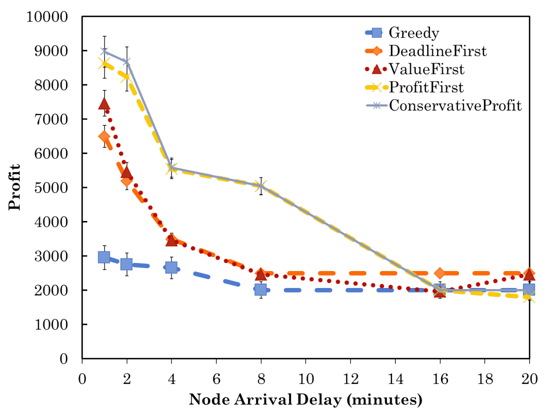


Figure 6.12: Impact of node arrival delay.

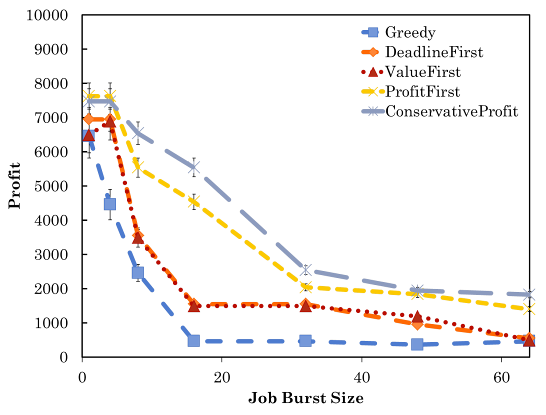


Figure 6.13: Impact of job burst size.

for all rental policies are fairly comparable. We therefore conclude that the cost-aware policies are somewhat ineffective in that they do not give additional benefit over other policies when the node arrival delay is too long. This reflects the importance of the rapid delivery of rented nodes from resource providers for more effective rental decisions.

6.4.4.6 Impact of Job Burst Size

We examine the profit sensitivity to burstiness in the workload. Most applications often generate and submits job in bursts as opposed to submitting individual jobs separated by a shorter period of idle time. In this experiment, we quantify the impact of profit sensitivity to burstiness as a function of job burst sizes. Larger burst sizes lead to more competition and subsequently more opportunity for ConservativeProfit, ProfitFirst, and ValueFirst to take advantage of cost information. In the experiments, the system load is fixed for different burst sizes.

The number of n jobs per burst is normally distributed with mean $n = (1 + M)/2$ where M is the upper bound of the number of requested nodes and the standard deviation $\sigma = n/4$. Each job has an exponential interarrival and an exponential task length. Further, the workloads are taken from a high ratio of urgent job distributions with a value factor of 4 and all jobs are also assigned with the same deadline factor of 4.

Figure 6.13 presents the results for all the rental policies under increasing job burst sizes. It can be observed that as the job burst size increases, the profits all fall substantially. This is due to a rapid increase in queue lengths when the job burst size increases. At smaller job burst sizes, high profits are attained since newly arriving jobs do not need to wait long in the queue because sufficient nodes are available to serve all queued jobs within their deadlines. However, as the burstiness increases, the number of queued jobs begins to increase and we can start to observe a widening variation in profits between rental policies. The ConservativeProfit gains the highest profit, while the ProfitFirst gains the second highest profit, which is followed by the ValueFirst. As expected, the Greedy policy earns the lowest profit with DeadlineFirst being the second lowest.

6.4.4.7 Impact of Job Sizes

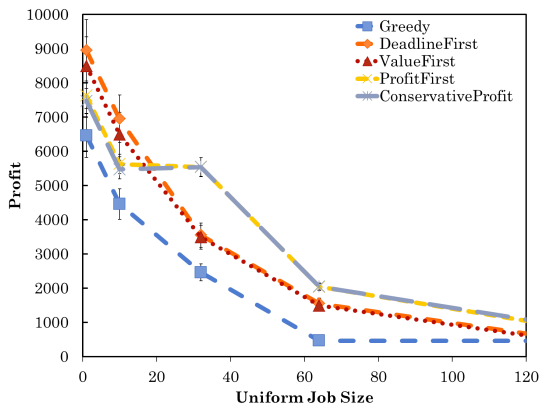
We examine the impact of job sizes on rental performance under different distribution job sizes. In the uniform distribution case, the number of requested nodes per job is uniformly distributed in the range of $[1...P]$ and the mean number of nodes per job is $n = (1 + P)/2$. Hence, the job size is uniformly distributed in the range of $[1...P]$ and the average number of requested nodes is $n = (1 + P)/2$. Alternatively, in the normal distribution case, we assume a ''bounded'' normal distribution for the number of requested nodes per job in the range of $[1...P]$ with mean $n = (1 + P)/2$ and standard deviation $\sigma = n/4$. Jobs in the uniform distribution case present larger variability in their degree of parallelism than jobs in the normal distribution case. In the normal distribution case, most of the jobs have a moderate degree of parallelism (close to the mean *n*). Since the distribution of job parallelism changes with time, for some time intervals, jobs have a highly variable degree of parallelism, while over other time intervals, the majority of the arriving jobs have moderate parallelism in comparison to the number of processors.

Figure 6.14 shows the profit as a function of increasing burst sizes for uniform and normal distribution cases. We can observe that profits decrease significantly as the number of requested *n* nodes increases under both uniform and normal distributions. In particular, the system profit falls significantly when job sizes are large. The decrease in profits can be attributed to the increased degree of competition that competing jobs face.

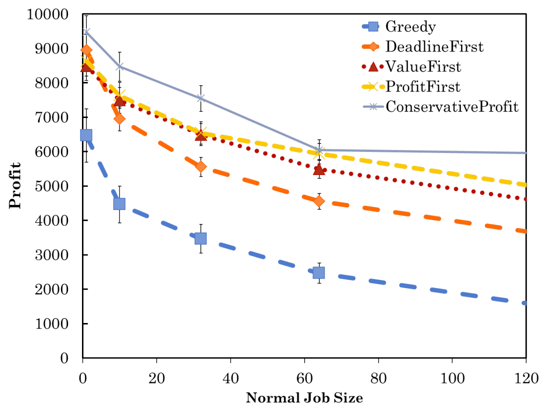
We can also observe that for uniform distribution, the fall in profits is even higher when compared to normal distribution. This shows the choice of policies not only depends on the overall system load, but also on the relative size of jobs in the system. When the variability of job size is high, we observe a significant reduction in profits for all policies due to higher variability of node demand. The system is unable to cope with the urgent demand of requests due to the high variability of job sizes. As a result, there are many jobs held in the queue waiting for nodes to become available.

6.4.4.8 Impact of Rental Cost

We examine the effect of increasing rental cost on profits. Figure 6.15 shows the impact on profitability when the rental cost per processor hour is varied. As the rental cost increases, our main observation is that the profits also decrease. As expected, the ConservativeProfit and ProfitFirst policies are shown to be susceptible under increasing rental cost because they both take into account profit and resource cost when making rental decisions. The ValueFirst policy also delivers higher profit than the Greedy



(a) Uniformly distributed job size.



(b) Normally distributed job size.

Figure 6.14: Impact of increasing job size.

and DeadlineFirst policies. As observed, the ConservativeProfit gains the highest profit, with ProfitFirst having the second highest. ValueFirst gains the third highest profit with only a slight difference in profit compared to the ProfitFirst. Finally, the Greedy policy incurs the lowest profit due to very high rental cost, and the DeadlineFirst policy incurs the next lowest profit.

6.5 Discussion and Comparison to Related Work

Despite the significant amount of research work on job scheduling in distributed systems (Feitelson et al., 1997; Zhang et al., 2000; Ernemann et al., 2002; Srinivasan et al., 2002), there appear to be very few research works that address algorithms and cost-aware rental policies under a dynamic resource environment.

Only a few computing centres reportedly (Gentzsch, 2001a; Beckman et al., 2006; Islam, Khanna, and Sadayappan, 2008) provide mechanisms that allow a user to be charged using a virtual currency for running his/her job on a resource infrastructure. In most cases, multiple queues for different levels of services are provided to determine the charges based on the resource used and quality of services sought. In such a model, the charge is generally proportional to the number of processors used and the job execution time. While such an approach may work well in a basic system, the resource management system that aims to offer QoS (i.e., deadline) guarantees cannot use such model (Islam et al., 2007).

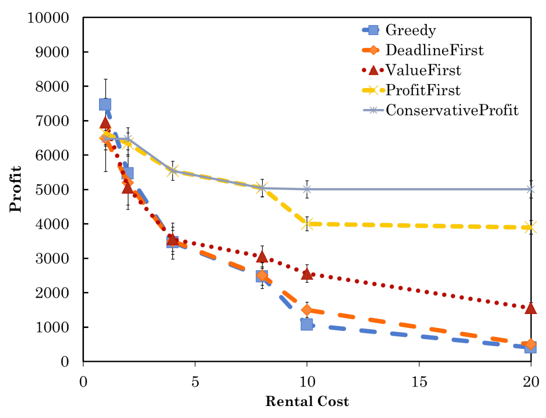


Figure 6.15: Impact of increasing rental cost.

There have also been some research studies (Chun and Culler, 2000; Chun and Culler, 2002a; Irwin, Grit, and Chase, 2004; Lee and Snavely, 2007) that provide similar approaches to our rental policies. Similar to our rental approach, they also propose economic-based scheduling approaches where a user can express his or her desire for a service through a virtual currency in order to differentiate users with different goals and preferences. For example, all the above mentioned studies recommend an 'aggregate utility' to measure the satisfaction of users with the resource allocation process. For job selection, the proposed scheme implements a heuristic approach where the job with a high value per unit running time gets preference. While their works are relevant, none of their studies have addressed the concept in the QoS-aware system where services like deadline guarantees and service differentiation are very important. Furthermore, their solutions do not consider a resource environment where nodes can be dynamically rented in response to changing resource demand. Therefore, their proposed algorithms do not explicitly consider resource cost as an important metric for the resource-provisioning process.

Similar to our work, Popovici and Wilkes (2005) consider a service provider that rents resources at a price. Developing this work, Burge, Ranganathan, and Wiener (2007) further propose several charging algorithms that consider both job revenues and resource cost when performing renting decisions. However, their approaches are slightly different because their algorithms focus on providing effective admission control in the event when resource availability is uncertain. On the other hand, we rightly assume that resource availability can often be guaranteed by the resource provider. Therefore, the focus of our rental policies is not on admission policies, rather we specifically provide a set of resource provisioning rules and strategies that can be employed by the system to meet the increasing demands of the applications' QoS requirements while keeping the resource cost at an acceptable level. Furthermore, previous work deal only with resource pools of fixed size whereby that the upper bound of rented nodes is fixed. For example, Popovici and Wilkes (2005) assume an environment where the resource pool size is limited to 20 nodes only. On the other hand, our rental policies relax this assumption so that additional resource nodes can be added dynamically in response to fluctuating resource demands.

To the best of our knowledge, there is no comprehensive costing model and framework for the evaluation of rental-based policies. Our work fills this gap by presenting a comprehensive costing model framework for the evaluation of rental-based policies. Based on the proposed costing model, we are able to introduce the ProfitFirst and ConservativeProfit policies that incorporate profit and net system revenue at runtime to guide rental decision making in a dynamic environment where the number of nodes are constantly changing. Our work also considers rental policies that take into account excessive rental cost: the ConservativeProfit considers the loss of profit from excessive rental cost when performing rental decisions. It aims to provide an adequate level of application satisfaction and keep the cost of renting to an acceptable level.

The economic-based approach also introduces many aspects that we need to consider, such as market dynamics, customer and resource bidding strategies etc. However, these aspects are beyond the scope of our current considerations. We deliberately assume that applications always provide true valuations of their jobs and that the VA is capable of charging users fairly. There is already a substantial body of research on overcoming the problems of charging and pricing resources accurately (Wolski et al., 2001b; Chunlin and Layuan, 2005; Buyya, Abramson, and Venugopal, 2005). For example, Libra (Sherwani et al., 2002) and Libra+$ (Yeo and Buyya, 2007) are common algorithms which have been proposed to make use of both monetary values and current system load condition to charge users for resource consumption. Chun and Culler (2002b) also build a resource management prototype that provides a market for time-shared CPU usage for various jobs. The most common economic-based model follows the auction-based resource allocation mechanism (Wolski et al., 2001a; Lai et al., 2004; Das and Grosu, 2005a) that has three major entities: users or buyers, resource providers or sellers and the resources to be sold. In this model, the user allocates the nodes for a specific duration and is willing to pay a certain value for the execution of the job. The resource provider sells the nodes to the user with an intention to maximize its overall profit. The auction process, which is generally proposed by the resource provider, considers the value or bid of all competing users and ultimately awards the resource to the highest bidder. Several auction-based mechanisms and policies have been proposed in distributed computing. Waldspurger et al. (1992) introduce the market-based microeconomic approach for batch scheduling. They utilize the auction process to choose the winner from the bids placed by different users. More recent work includes that of Chun et al. (2005) who propose an auction-based microeconomic approach for parallel job scheduling. In their scheme, test-bed resources are allocated using a repeated combinatorial auction within a closed virtual currency environment. Users compete for test-bed resources by submitting bids which specify resource combinations of interest in space/time along with a maximum value amount the user is willing to pay. A combinatorial auction is then periodically run to determine the winning bids based on supply and demand while maximizing the aggregate utility value delivered to users. Their work has shown that the support of a computational economy can lead to self-regulated accountability in distributed computing.

Finally, our work assumes that resources are always available in times of high demand from external resource providers. We realize that this is a strong assumption, but this assumption is realistic given the high availability of resources from todays Cluster, Grid, and Cloud systems.

6.6 Chapter Summary

In this chapter, we explored the effects of different policies that take into account job monetary values, job deadlines, system revenue and system profitability, and we examined how load, job mix, job values, job deadlines, node heterogeneity, node arrival delay, job sizes, and rental cost influenced the system's profit. We proposed several cost-aware policies that make use of individual job information to increase application satisfaction and to maximize the system's profitability under varying workloads. All these policies have been extensively evaluated to compare their advantages and disadvantages across different application and the deployment of new nodes from the resource providers, and/or on parameters and system parameters using simulations. Our experimental results provide an insight into the effectiveness of the proposed rental policies:

1. The *aggressive rent+conservative release* policy generally performs better than the other rent and release policies because it optimistically rents more nodes to serve queued jobs and at the same time it does not incur a high rental cost. From the poor performance of both the *aggressive rent+aggressive release* and the *conservative rent+aggressive release* policies, we are able to conclude that the aggressive release policy is not suitable for all workload conditions. The conservative release policy is recommended instead for all workload situations (at any percentage of urgent requests).

2. Both ProfitFirst and ConservativeProfit policies consistently produce the highest profits with lower penalty costs than other policies. They both consistently adapt to varying load, job mix, job values, job deadlines, node heterogeneities, arrival time, and rental cost better than any other policies. This is because they both take into account system profitability each time resource allocations and rental decisions are carried out. We have also observed that the ConservativeProfit policy performs slightly better than the ProfitFirst policy in all cases. This is because it is less aggressive when renting additional nodes. This has the positive effect of lowering the rental cost while not incurring too much penalty for queued jobs. On the other hand, the ProfitFirst policy rents additional nodes as long as the jobs are profitable, but this has the somewhat perverse effect of incurring a very high rental cost.

We conclude that (i) the proposed rental-based policies were shown to provide significant benefits over a static policy; (ii) there is encouraging initial evidence that by combining the information on job monetary value, job deadline and system net profit on the fly (at runtime) when making rental decisions leads to higher overall profit increase; and (iii) there is good evidence to recommend the use of either the cost-aware ProfitFirst or the ConservativeProfit policy to increase system profitability and maximize application satisfactions for varying workloads and system parameters.

This chapter has developed and presented policies that can be used as the foundation for improving productivity, managing costs, and improving return on investment for renting resources on Cluster or Grid systems. The proposed policies also attempt to provide the mechanism for the VA to balance the cost of acquiring computational resources and the cost of satisfying application QoS requirements. Our contributions include extensive evaluation of a new family of cost-aware rental policies under varying workload characteristics using real workload traces collected from the Lawrence Livermore National Lab (LLNL) Thunder workload log.

With our experiments, we demonstrated that the cost-aware rental policies provide significant performance and cost gains over a static policy. However, in several observations, we have discovered that profits obtained from our rental policies are severely affected by several factors which include the node arrival delay, job burst sizes (interarrival rates), job sizes (degree of parallelism), and resource cost. Such factors could have an adverse effect on the execution of adaptive and interactive distributed/parallel applications that are very dynamic in nature whereby their computations can vary significantly during runtime. Therefore, rental policies that only react to current workload conditions alone may not be effective because the system does not have information on the overall application workload characteristics. Considering this issue, it is envisaged that an additional control is needed to address this limitation, and this is the subject of the next chapter.

Chapter 7

Service Level Agreements

7.1 Introduction

In Chapter 6, we have proposed and evaluated several rental policies by extensive simulation with the aim of increasing application quality-of-service (QoS) requirements and reducing resource cost. However, employing rental policies alone may not be efficient to satisfy the QoS requirements (i.e., deadlines) of applications which have highly variable and chaotic workload. This is because the information provided by an individual job does not capture the overall application workload characteristics.

Instead, it would be beneficial for the system to obtain information concerning the overall application workload characteristics so that violation cost (penalty cost) and resource cost can be further optimized. We need mechanisms to enable the system to plan for the future, by determining the amount of nodes to rent to accommodate near future demand. In order to achieve this, there must be some form of knowledge available to the VA system in predicting future demand. It is envisaged that Service Level Agreements (SLAs) can provide the effective means to do so. An SLA provides a powerful instrument for describing all expectations and obligations in the business/commitment relationship between two or more separate entities (Leff, Rayfield, and Dias, 2003). The process of SLA negotiation differs significantly from the regular job submission interface to the resource management system.

Table 7.1 summarizes the most significant differences between standard job scheduling and SLA-aware scheduling. Conventionally, a user submits a job description along with the tasks, and it is scheduled in accordance to its job description at the point of time when the job arrives to the system. Since a job is scheduled individually, there is a possibility that a job might be rejected by the system when there are insufficient resources to run the job (Yeo and Buyya, 2007; Burge, Ranganathan, and Wiener, 2007). Furthermore, even if a job is accepted by the system, there is no guarantee that the quality-of-service of the job can be met under very high load and very low node availability (Popovici and Wilkes, 2005). However, with SLAs, these issues can be prevented by allowing users to specify resource requirements for their present and future computational needs (Sakellariou and Yarmolenko, 2008). This would also give the opportunity for the system to plan resource allocation more effectively.

SLAs can also minimize the overhead of having to negotiate resources each time when there is a job request. With a SLA, once a SLA request has been accepted, the both VAs and resource providers must adhere to the contractual agreements. Therefore, additional nodes can be requested without negotiation

	Job Descriptions	Service Level Agreements
Scheduling time frame	Present.	Present and future.
Admission policy	Jobs can be rejected due to load variability.	All jobs can be accepted to maximize revenue.
Quality-of-service	Unlikely to meet under very high load and very low node availability.	Rent sufficient nodes for each to meet all current and future QoS.
Negotiations	Tedious because both user and system must agree on the monetary value for each newly arrived job.	Additional resource nodes can be requested without negotiation overhead during application execution.
Rental policy	Rents resource nodes based on present load and node availability.	Rents resource nodes based on all accepted SLAs.

Table 7.1: Comparison of standard job scheduling that uses job description information and SLAs that make use of high-level application information.

overhead during application execution (Pichot et al., 2009).

Unlike individual job descriptions, SLAs may include include high-level information such as the total overall load that the application can impose, the total application running time, and the total offered monetary value for serving the application (AuYoung et al., 2006). Therefore, rental decisions can be made based on all current accepted SLAs. SLAs also include a penalty agreement for poor performance or for when there is an agreement violation: how much the system should be penalized for not meeting specific QoS requirements, e.g., if the response time is too long for many jobs or if too many deadlines are missed (Islam et al., 2007). Given many of the advantages, it is advocated that SLAs can be effectively used to provide a high guarantee of job completion (i.e., high QoS satisfaction).

7.1.1 Incorporating SLA into Resource Allocation

The vision of SLA based scheduling assumes that the SLAs themselves are machine readable and understandable (Hudert, Ludwig, and Wirtz, 2009). This implies that any agreements, between the parties concerned, for a particular level of service need to be expressed in a commonly understood (and legally binding) language. There has been early work on generic languages for SLA description (Jennings et al., 2001; Czajkowski et al., 2002), but none related to the particular problem of the requirements associated with job submission and execution on high performance computing resources.

In essence, some commonly agreed protocol needs to be followed during the negotiation process. This protocol needs to take into account negotiation over geographically distributed resources and networks (e.g., what if the system is unable to serve job requests due to high rental cost?), and how the system abides with appropriate legal requirements (e.g., what is the process of paying out compensation to the user due to SLA violation?). Furthermore, during negotiation, a virtual authority (VA) should be able to reason about whether a contract is acceptable in relation to its offer and possibly it should be able to make counter-offers (Green et al., 2007). There has been a significant amount of work on these topics. However, current research on SLA negotiation is very much related to SLA web service specification (Stuer, Sunderam, and Broeckhove, 2005; Green et al., 2007; Litke et al., 2008) and networking (Czajkowski et al., 2002; Duan, Zhang, and Hou, 2003) rather than the problem of using SLAs for job scheduling.

Furthermore, the current approach for the scheduling of jobs on high-performance computer resources is based on advance reservations (Foster et al., 1999a; Smith, Foster, and Taylor, 2000) (with the possible addition of backfilling techniques (Mu'alem and Feitelson, 2001; Zhang et al., 2001)). As we have stated many times in the previous chapters, such an advance reservation approach is very ineffective due to job execution uncertainty and the lack of control. Therefore, the use of SLAs would be of paramount importance to promote long-term planning, and to enable the system to plan renting decisions more effectively. However, this would require the development of a new set of SLA-based algorithms and policies for effective scheduling and rental decision-making. In this chapter, we propose an SLA management (SLAM) framework that offers such a capability and we then propose and evaluate several SLA-aware policies that can be employed within the framework.

7.2 Framework

In this section, we present the design framework of the SLA Management (SLAM) system. The SLAM architecture is shown in Figure 7.1, which is an extended framework originally proposed in Chapter 4.

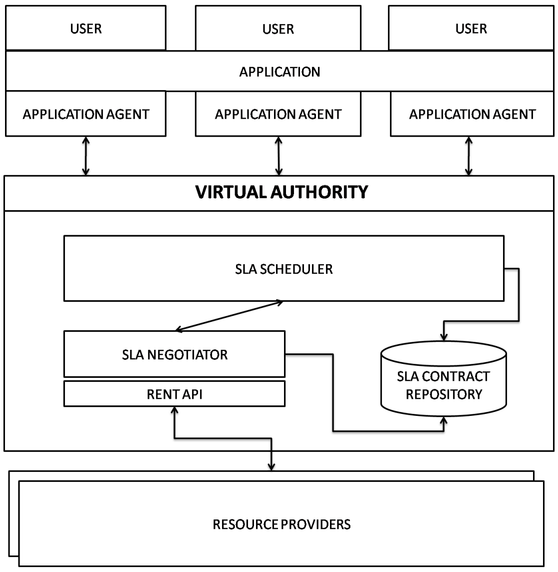


Figure 7.1: Service Level Agreement Management framework (SLAM).

The core components of the SLAM framework are the SLA Scheduler and SLA Negotiator. The SLA Scheduler is effectively in charge of serving applications' job requests. It decides whether to accept or to reject a SLA contract based on the QoS parameters provided by the user or by current state of the system. If it accepts the SLA contract, it stores the contract in the SLA Contract Repository for future retrieval and reference. The SLA Contract Repository stores all the accepted contracts from users so that when a new job arrives, the system can effectively use the updated information, e.g., remaining jobs, total value etc. using the information from the stored contracts in order to guide renting decisions.

Hence, when a new job arrives, the SLA Scheduler retrieves the information from the SLA contract repository and uses the information provided by the SLA contract and the individual job to determine how many resource nodes to rent and when exactly to rent them. The main objective is to maximize the revenue accrued from charging the users: by maintaining high application QoS satisfaction and by lowering the cost of resource consumption. High application QoS satisfaction is measured by how well the system completes users' jobs within users' deadlines. The low cost of resource consumption is measured by the ability of the system to reduce the associated cost of renting nodes when serving users' jobs.

Without a SLA mechanism, the system cannot perform long-term planning because it can only rely on an individual job's information to differentiate the priority execution of the different jobs. For illustration, consider a case where there are two jobs and one processor in the system: a job j_a is assigned a monetary value of 10, whereas a job j_b is assigned a monetary value of 20.

In the standard mechanism, job j_b should be given a higher priority for execution because its value is higher. However, if the SLA contract of j_b indicates that its remaining load (runtime) is less than 10 seconds while the remaining load (runtime) of job j_a is approximately one minute (~60 seconds), it may then be more appropriate to execute job j_a first and execute job j_b after j_a has completed its execution. Higher revenue can be gained this way because there is high utilization of resources (since $max(10 * 60, 20 * 10)$).

However, if job j_b has a very strict deadline of 30 seconds while a job j_a has a deadline of 60 seconds, it is not possible to serve job j_a with a deadline of 60 seconds without violating the deadline of job j_b since there is only one available processor. Therefore, in such a situation, the system must use this knowledge to rent additional resources from external resource providers so that both jobs can be accommodated within their QoS requirements.

As noted earlier, the SLA Negotiator is responsible for the negotiation of rentals from the external resource providers. The SLA Negotiator negotiates resource nodes from external resource providers in response to the resource demand. Based on the information from the SLA contract, the SLA Negotiator rents nodes in response to dynamic variations in load balance and unpredictable job arrivals. It would have the ability to observe the occurrence of jobs in the system in order to recognize patterns among such jobs, and rent extra nodes accordingly. This ensures sufficient nodes are always maintained to accommodate present jobs and future jobs that are expected to arrive in the near future.

7.2.1 SLA Parameters

As discussed above, individual job information alone is not sufficient for the VA to perform long-term planning. The system needs to retrieve additional information on the overall behavioural characteristic of the application in order to maintain high QoS satisfaction and to improve overall application performance. The basic idea of the SLA approach is to use the SLA contract information to predict resource demand more accurately. *However, what kind of information should be incorporated within a SLA contract?*

In our framework, it is proposed that a SLA contract contain and include the following information and parameters:

- Duration of contract: the total duration (timeframe) which the user wishes to complete all jobs;
- Total number of jobs: estimated total (aggregated) number of jobs which are going to be submitted.
- Total estimated number of required nodes: estimated total number of nodes required for the whole

application execution.

- Estimated average number of requested nodes: degree of parallelism (average number of nodes requested for each job) and the standard deviation σ_S of job size *S*;
- Estimated average job deadline: estimated average job deadline *D* and its standard deviation σ_D;
- Estimated average penalty for non-compliance: zero penalty implies best-effort jobs (jobs with no specific deadline requirements) while penalty $\delta_j > 0$ indicates jobs with deadline requirements.
- Total monetary value (TMV) for the contract: the accumulated total offered value of all jobs from the start of job submission until the end of job submission.

The user should specify a SLA contract with the above parameters, but not the precise timings of when jobs will arrive, or their job individual sizes. This is because we cannot always realistically anticipate that individual job parameters are available in advance due to the unpredictable nature of job execution. Therefore, users are given the flexibility to specify standard deviations for both job size and average job deadlines. However, for simplicity, we will assume a somewhat simplified model in specifying a distribution.

The TMV for the contract is the total offered value by the user for all jobs from the submission of first job until the submission of last job ($\sum_{j=1}^{J} V_j$) where *J* is the total number of jobs in the system. Based on the TMV, the user may appropriately assign a monetary value to each job as long as the accumulated monetary value for all assigned jobs do not exceed the TMV. This approach prevents users from assigning high monetary values to non-urgent jobs. Most importantly, this approach encourages users to provide an accurate monetary value for each job. As noted in the previous chapter, we make the assumption that users are capable of differentiating the priorities of different jobs. Without such a capability, the applicability of the SLA-aware policies would be limited.

In future work, users may be offered financial compensation in return for the additional information provided. For example, users may also specify and include additional information such as the expected arrival rates (how often tasks arrive in the future) and the expected deadline ratio of job arrivals (ratio of urgent and non-urgent jobs). When users are financially rewarded for such information, this will encourage them to provide more information about its workload. Consequently, this would also increase the ability of the system to enhance renting decisions. As the end result, this reduces the chance of users being dissatisfied and disappointed.

7.2.2 Submission and Charging Model

Figure 7.2 illustrates the SLA submission model in SLAM. First, when a user submits a SLA contract, the request is forwarded to the HAdmission service to determine whether or not to accept the SLA contract. The decision will be made based on (i) TMV of the contract and (ii) current and future resource availability[1]. For the purposes of our evaluation, we assume that the contract will be accepted regardless of the contract's TMV value (we assume that no price feedback or admission control is considered). We

[1]HRental anticipates whether sufficient resources are available within the contract duration.

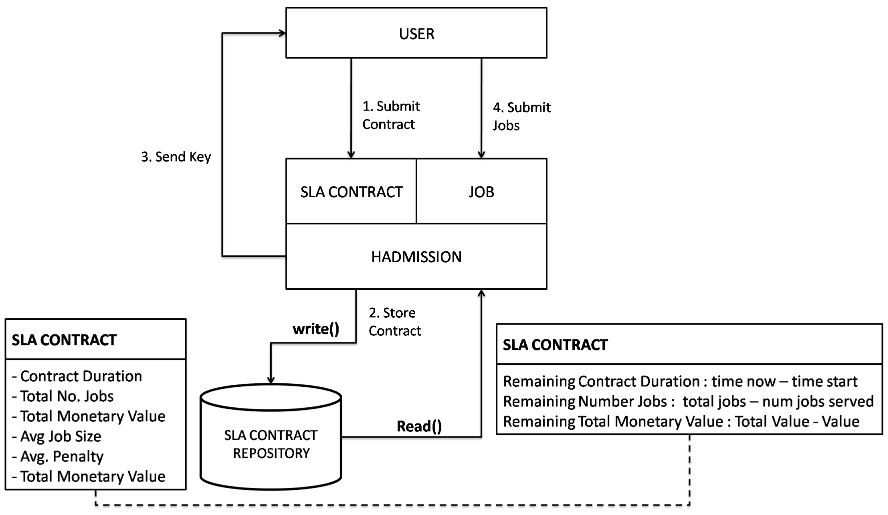


Figure 7.2: Job submission model in SLAM.

also assume that the resource availability at future times is guaranteed due to the high availability of resources worldwide. Once a contract has been accepted, it is recorded and stored in the SLA Contract Repository. The user is given a unique key for identification upon the acceptance of the contract.

After successful acceptance of a SLA contract, the user can start submitting the first job. The user submits a job along with the job description and its given identification key. Based on the identification key, the HAdmission verifies whether the submitted job matches any of the SLA contracts from the SLA Contract Repository. If verification fails, the job is rejected and a rejection notification is sent to the user. If verification succeeds, the system traverses the repository to identify and locate a matching SLA record. If a record is found, the system obtains the necessary information regarding the contract such as its remaining load, the remaining number of jobs, the estimated average deadline, the estimated average penalty cost, and the remaining TMV.

Figure 7.3 further illustrates the charging model of the SLAM framework. Upon job arrival, the HAdmission verifies the job to check whether the SLA contract which corresponds to the job exists in the SLA Contract Repository. Based on the information provided from the corresponding contract as well as the individual job information, the HAllocator performs the appropriate scheduling and renting decisions.

The SLA Contract Repository stores both the revenue and expenses information. The revenue contains information on deployment overhead cost and resource node usage fee including penalty cost. When a job *j* is being served with its requested number of nodes, the user is charged immediately and the revenue is recorded in the SLA Contract Repository. The HPricer service is responsible for recording the earning revenue V_j of job *j* information in the SLA Contract Repository. Further, if there is any penalty cost δ_j incurred during application execution, the revenue V_j is also updated once again in the SLA Contract Repository at the end of each job completion.

The expenses information represents the rental cost which includes the total cost that the system has already spent on rented nodes. The HRental service takes care of the rental action. Each time the HRental service rents a resource node, the HAccount service is triggered to update the expenses information (i.e., rental cost) to the Repository. Similarly, at rental termination, the HAccount service is also triggered to update any additional overhead and usage fees that have been incurred during job execution.

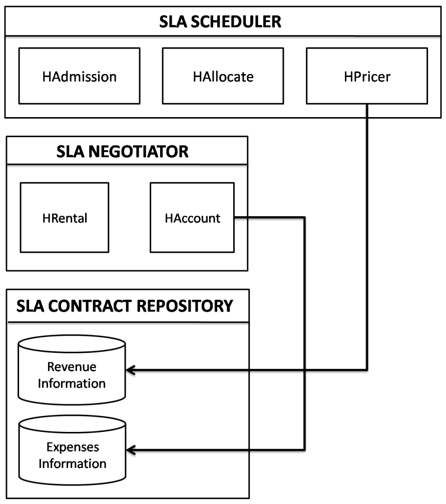


Figure 7.3: Charging framework in SLAM.

7.2.3 Penalty

The SLA may include penalty clauses that penalize the VA if it fails to execute the jobs within the QoS constraints or requirements. The existence of SLA penalty clauses enables the system to maintain certain standards of services expected by users. In the event that the VA fails to meet a job's deadline, a compensation (penalty) is paid out to mitigate the losses incurred by the user.

A penalty will reduce system earnings. In such cases, the earnings must be adjusted due to the lost revenue from compensation. Figure 7.4 illustrates how the SLAM system adjusts the level of revenue from the SLA Contract Repository when there is an SLA violation. Initially, when nodes are being allocated to a job request, the HPricer service transfers the job monetary value from the TMV Repository and records the increase in profit in the SLA Contract Repository. Optionally, the AA may also deploy the HMonitor service to periodically monitor and record the usage time between the start of job execution until job completion. This log record can be useful later for accounting purposes in the event of a dispute between the user and the VA system. In the event that a job's deadline has been violated, the HAccount service deducts the profit from the SLA Contract Repository based on the agreed penalty cost as specified in the SLA. Finally, the HPricer service transfers the compensation fee (penalty cost) to the TMV Repository.

7.3 Experimental Methodology

A set of experiments have been carried out according to the SLAM framework. The aim of the experiments is twofold: first to validate that the SLAM framework enables effective renting decisions and second to compare the efficiency of the proposed SLA-aware policy options.

This section describes the experimental setup and the default parameters used in the experiments. The default parameter settings are shown in Table 7.2. For the experiment, each AA generates a job

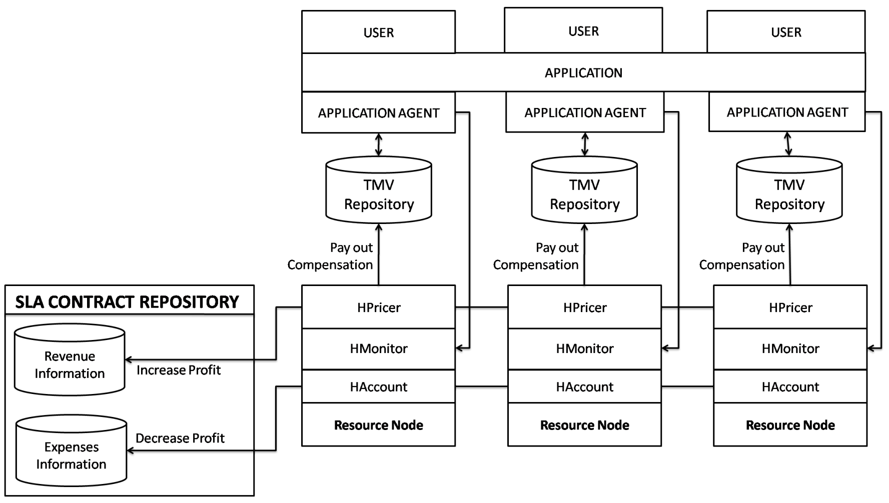


Figure 7.4: Penalty handling in SLAM.

request based on the workload traces of the LLNL Thunder cluster used in Chapter 6. As noted in the previous chapter, the LLNL workload traces were chosen because the workloads contain a large number of smaller to medium jobs which represent the highly variable and chaotic job submissions of distributed and parallel applications.

None of the workloads available contains any information pertaining to how SLA contracts are generated. Therefore, the SLA parameters must be chosen judiciously. For the experiment, the system initially consists of U users and each user generates a contract and sends it to the system with an exponentially-distributed interarrival time. Upon the submission of a contract, the user waits until they receive a notification from the system that the contract has been accepted. Upon contract acceptance, the user's application starts submitting batches of sequential and/or parallel jobs based on the workload traces.

Since the traces do not contain information pertaining to job urgency or the monetary amount the application is willing to pay for job execution, we follow a similar methodology as in the previous chapter to model these parameters through two job classes: (i) high urgency and (ii) low urgency. The workloads consist of *X%* urgent jobs and *100-X%* non urgent jobs.

For the sake of brevity, the deadline factor between urgent and non-urgent jobs is stretched by a deadline factor *df*. To examine the impact of SLA-aware policies under strict and flexible requirements, we evaluate the impact of cost under various deadline factors. The monetary value for each job is categorized based on the urgency. For a non-urgent job, a low monetary value is assigned using a normal distribution with mean 10 and standard deviation of 2. On the other hand, a high monetary value is assigned using a normal distribution with mean 40 and standard deviation of 8. In this example, the difference between low and high monetary value is a factor of 4. A mean monetary value factor of 4 is chosen based on previous related work (Chun and Culler, 2002b; Yeo and Buyya, 2007). This model is realistic since an application that submits a more urgent job with a strict deadline is more likely to offer a higher monetary value for the job.

The TMV for each SLA contract is calculated from the total number of generated jobs and their urgency. For each contract, the total number of K jobs is generated and each job is assigned as urgent or non-urgent according to the fraction of urgent and non-urgent requests in the workload. Further, the TMV is computed by summing up all jobs' monetary values of the contract ($\sum_{j=1}^{J} V_j$).

For each experiment, we assume that the SLA Scheduler periodically schedules jobs in the queue every 0.1 hours. We also assume that the SLA Negotiator periodically reviews the amount of additional nodes to rent every 0.5 hour. The periodic interval for renting decisions (periodic rental interval) is set higher than the periodic interval for scheduling decisions (periodic scheduling interval) in order to avoid over-optimistic renting. The SLA contract duration follows a gamma distribution with mean 10 hours and standard deviation of 2 hours. Gamma distributions are chosen because they behave roughly like normal distributions, but have the attractive property that they do not generate negative values. Using a normal distribution and suppressing such values would result in a new, not-quite-normal distribution with a slightly different mean than intended.

Parameter	Default Value
Simulation length	100 hours
Runs per data point	20
Contract inter-arrival time	exponential(1.0) hour
SLA contract duration	gamma(10.0,2.0) hours
Periodic scheduling period	0.1 hour
Job interarrival time	0.15 hour
Periodic rental period	0.5 hour
Rental duration	1.0 hour
Slack and load threshold	0.5
Urgent:non-urgent ratio	80:20
Penalty value	-job monetary value
Low:high job value factor	4
Urgent:non-urgent factor	4
Rental cost factor	4
Cost of a node per hour	0.26

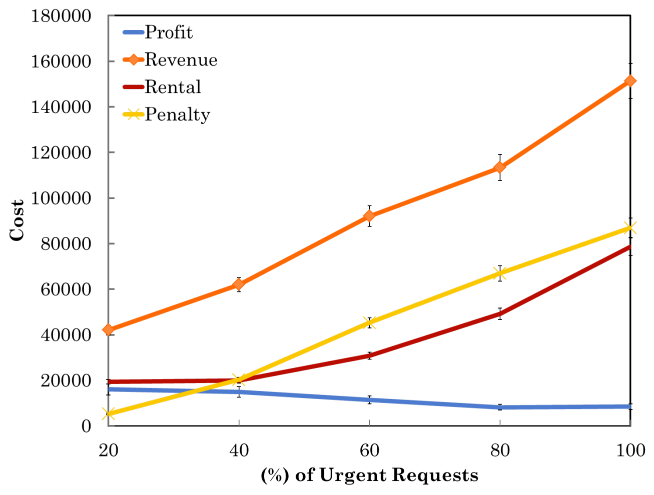
Table 7.2: Default simulator parameter-settings. The notation *distribution(x,y)* means a distribution of the given type with a mean of x and a standard deviation of y.

7.4 Non SLA-aware Policy

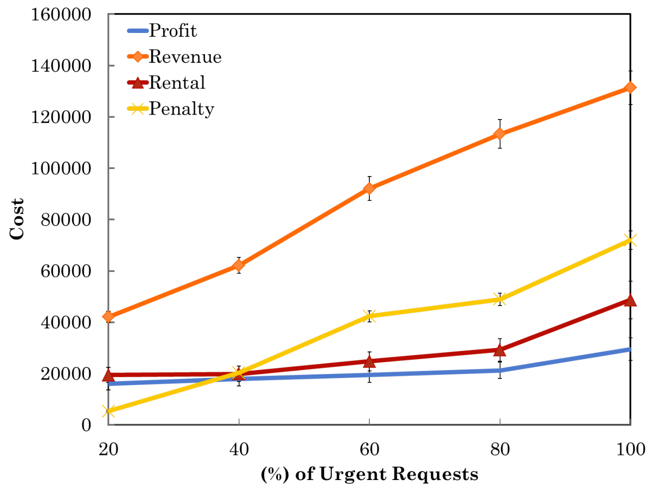
Before we evaluate how SLA contracts can be effectively used to guide scheduling and renting decisions, we first present simulation results for the non SLA-aware policy so that meaningful comparisons can be made with SLA-aware policies.

7.4.1 Non SLA Heuristic

We first examine a simple non SLA-aware Greedy policy that evaluates renting decisions based on the current load of the system. The Greedy policy is based on the aggressive rent and conservative release policy which has been described in detail earlier in Section 6.3.1. Recall that with the Greedy policy, the system rents sufficient nodes for each queued job as soon a rental threshold occurs. The aim is to fullfil as many job requests as possible. The Greedy policy does not make use of application QoS parameters such as job deadline and job monetary value information.



(a) strict *df*=4.



(b) flexible *df*=8.

Figure 7.5: Greedy Heuristic scheme under varying fraction of urgent requests for *df*=4 and *df*=8.

Figure 7.5 shows the cost breakdown the Greedy heuristic scheme under strict and flexible deadline conditions. The cost includes profit, revenue, penalty cost and rental cost. Our first observation is that the penalty cost is significantly high. Although we can see that the revenue increases steadily as more jobs arrive in the system, interestingly, the rate of increase in penalty cost is almost comparable to the rate of increase of the revenue. We can also observe that the rental cost increases as the fraction of urgent requests increases. This shows that even when the system continues renting additional nodes in order to satisfy the incoming job requests, it is still unable to increase its overall profit – the profit is substantially low. This observation illustrates that the non SLA-aware Greedy policy is somewhat ineffective because it has to rent quite a significant amount of nodes in order to fulfil the QoS of the incoming job requests.

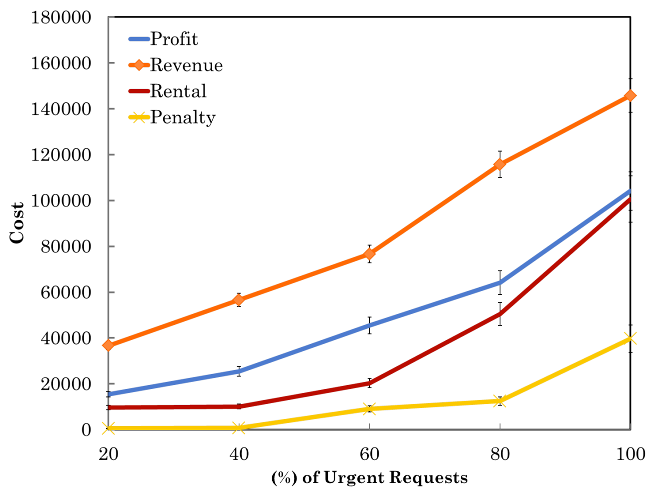
Moreover, we can observe that the increase in the number of rented nodes does not have a positive effect at all in reducing the penalty cost. As observed, while the rental cost steadily increases, the system fails to reduce the penalty cost. It is clear that the Greedy heuristic excessively rents too many nodes without providing any cost benefits to the system.

Recognizing that not every best-effort and urgent request requires equal responsiveness, the system should adjust its scheduling and renting decisions based on job priorities (e.g., urgent and non-urgent jobs). However, it is often very difficult to make optimal (or suboptimal) scheduling and renting decisions without any knowledge of future application behaviour. Even in conventional Cluster or Grid systems, it is compulsory for the users to specify estimated job runtime when submitting hard deadline (urgent) jobs (Islam et al., 2003). Such systems support advance reservation to provide QoS support. However, if the actual job runtime exceeds user runtime estimates, the job has to be terminated immediately and needs to be resubmitted again (F.Heine et al., 2005). Therefore, users are forced to provide exact estimates of job runtimes. This is often not feasible because users cannot anticipate the job runtimes in advance due to the unpredictable processing demands of distributed/parallel applications.

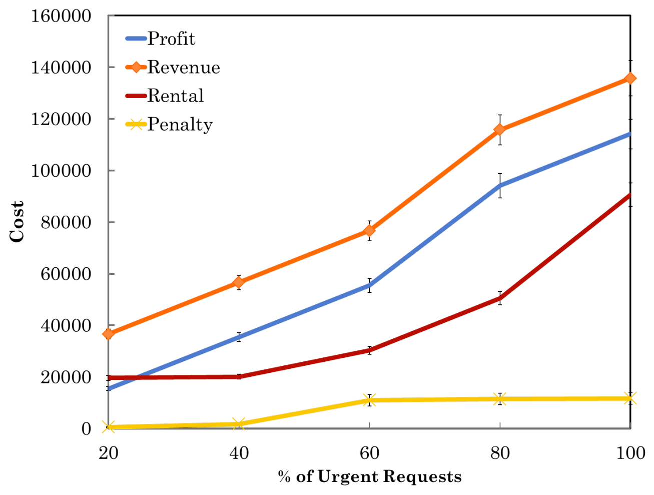
It is envisaged that if the system could effectively predict application future load reasonably well, resource nodes could be rented periodically within pre-specified intervals in order to maintain sufficient nodes for serving current jobs and future arrival jobs. If the system could somehow enforce applications to quantify some information about their future behaviours, it could then make use of the information provided to anticipate future demand. This prompts the development of SLA contract-based scheduling and renting policies, that we evaluate next.

7.5 SLA-aware Policies

The previous Greedy policy performs renting decisions based on information provided from the current information in the system queue. The policy assumes that no information on the overall application execution characteristics is available. In the next section, we present and evaluate several SLA-aware rental policies that utilize information of future application execution characteristics which are expressed using SLAs.



(a) *df=4*.



(b) *df=8*.

Figure 7.6: Rigid heuristic scheme under varying fraction of urgent requests for *df=4* and *df=8*.

7.5.1 SLA Rigid

We examine a simple rental policy that evaluates renting decisions based on exact estimates of demand. The basic idea of the policy is to rent exactly N nodes for the period covered in each SLA contract. For example, if the user specifies D as a contract duration and M as the total estimated number of required nodes, the system then issues a single rental request to rent M nodes for D rental duration.

Figure 7.6 shows the cost breakdown for the rigid heuristic scheme. Comparing Figure 7.5 and Figure 7.6, we can observe the improvement in profit and penalty cost for all fractions of urgent requests. This result is attributed to the priority queue employed that always rents nodes exactly based on information provided in the SLA contract. This improvement is especially significant because we can observe that the penalty cost is now lower than the rental cost. Furthermore, when comparing the Greedy heuristic scheme with the rigid heuristic scheme, the rigid heuristic scheme improves the overall profit for the system at any fraction of urgent requests. As observed, when the fraction of urgent requests increases, we can also see that the profit improvement increases.

However, one interesting observation is in respect of the increased rate of the rental cost for SLA Rigid Policy. As observed, as the fraction of urgent requests increases, the rental cost also increases significantly. This result is attributed to the rigid strategy that serves all queued job requests without optimizing rental cost. It is anticipated that if the system could effectively predict future load reasonably well for the next rental period, less nodes need to be rented to accommodate QoS (i.e., deadlines) requirements of current and future jobs.

7.5.2 SLA Load-aware

We consider a SLA-aware load policy that makes use of the estimated load information provided by the application in the form of SLA contracts to guide scheduling and renting decisions.

The basic idea of the scheme is to rent nodes for queued jobs based on the application's anticipated load over a specific time interval. If the system is able to predict future system load with reasonable accuracy, it will be able to rent the least amount of nodes required to accommodate the job requests. This avoids excessive rental cost that could be incurred due to over-optimistic renting.

Using a SLA contract, the user provides information regarding the application workload such as the total load (processing units) of its overall execution and/or an estimated number of jobs that will be generated throughout the lifetime of application execution. In this subsection, we shall consider a case where a SLA contract is submitted to the system during job submission with the following information: (i) the application total load and (ii) the estimated total number of jobs. We aim to examine how such information can be effectively used by the system to predict the future demand of the application.

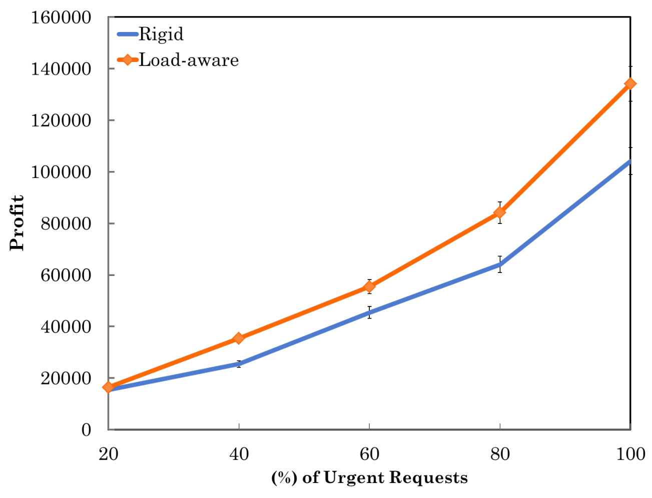
As noted earlier, the SLA contract must be accepted before the application can start sending any of its job requests. Upon accepting a SLA contract, the system identifies the amount of processing loads and the estimated number of jobs specified within the contract. Whenever a job j of a contract *sla* is scheduled by the system, the system computes the remaining load $reml$ for all queued jobs that belong to contract *sla* and remaining number of jobs $remj$ of a contract sla as follows:

$$reml_{sla} = \frac{reml_{sla} - \sum_{j=1}^{sla} l_j}{otl} \tag{7.1}$$

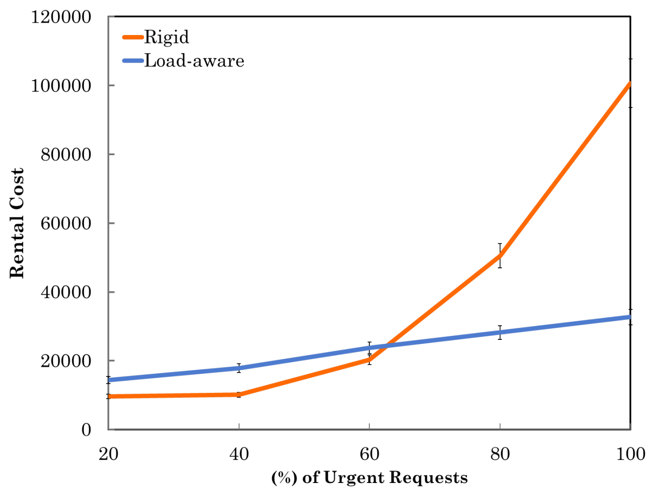
$$remj_{sla} = remj_{sla} - 1 \tag{7.2}$$

where l_j is the job's j estimated load and *otl* is the original total load of the contract when it is first initially defined within the SLA contract. Effectively, the remaining load and remaining number of jobs for a contract sla is computed using information from both the SLA contract and the individual job's information. When the threshold has been triggered, the system traverses each queued job that belongs to the contract sla and checks whether the remaining *sla* has fallen below the specific load threshold. If the remaining load $reml_{sla}$ has fallen below the threshold, the system immediately rents the required S_j nodes for each queued job that belongs to the contract sla.

The strategy is to rent extra nodes only for queued job *j* that belong to the *sla* contract with lower remaining load $reml_{sla}$. The idea is to recognize the benefits of renting extra nodes in accommodating



(a) SLA Rigid vs. SLA Load-aware.



(b) SLA Load-aware heuristic.

Figure 7.7: Cost Comparison of SLA Load-aware policy versus SLA Rigid scheme.

a job with substantial load. The benefits derive from potential future revenue which is expected to be generated from executing job j of that specific sla contract.

In Figure 7.7a, we compare the cost breakdown of the SLA Load-aware policy with the SLA Rigid policy. The remaining load threshold is set to 0.5. Our first observation is that the profits for both policies are comparable when the system is loaded with a few urgent jobs. At a lower fraction of urgent requests, both policies are able to maintain a high level of profit. However, as the fraction of urgent requests increases (up to 60% of urgent requests), we start seeing the effectiveness of the SLA Load-aware policy in generating higher profit compared to the SLA Rigid policy. We can also observe that the profit for the SLA Rigid policy starts to decline. The result is attributed to the incompetent decision-making of the SLA Rigid policy in renting too many nodes due to sudden arrivals of urgent jobs in the system.

Figure 7.7b further shows the rental cost for both SLA Load-aware and SLA Rigid policies under varying percentage of urgent requests. We can see the rental cost for the SLA Rigid policy increases substantially as the fraction of urgent requests increases. As a result, this significantly reduces the overall profit. However, the same does not apply to the SLA Load-aware policy because the rental cost can be maintained steadily when the fraction of urgent requests increases. This explains why the profit for the SLA Load-aware policy continues to improve as the fraction of urgent jobs increases. In particular, from Figure 7.7a, we can see that at high load (above 60% urgent requests), the SLA Load-aware policy generates approximately 40% higher profit when compared to the SLA Rigid policy. The profit continues to increase as the fraction of urgent requests increases.

Unlike the SLA Load-aware policy, the SLA Rigid policy is unable to maintain the system's rental cost because there is no mechanism to evaluate the remaining load and remaining number of jobs for individual applications. This shows that predicting future demand based on the remaining load information provided by the SLA contract and the load information concerning individual jobs can be effectively used to improve the overall user satisfaction and increase the profit margin of the system. Since the system is able to anticipate future workload in advance, jobs can be scheduled more efficiently and nodes can be rented with minimal unused capacity. We can also observe that the benefits of enforcing SLA information are most pronounced when the workload consists of a higher fraction of urgent requests.

7.5.3 SLA Value-aware

In the previous policy, no attempt has been made to ensure that urgent jobs can be satisfied within deadline constraints. Although the policy works well under a higher fraction of urgent requests, there is no QoS support for urgent jobs that require their deadlines to be satisfied.

In this subsection, we propose the SLA Value-aware policy that attempts to enhance the profit by incorporating knowledge of job monetary values. In such a policy, the user specifies exactly the total amount they are willing to pay for executing their jobs within the SLA contract. The information on the total (aggregated) monetary values provided by the user within the SLA contract will be used effectively to guide scheduling and renting decisions.

The basic strategy is to maximize the revenue by differentiating high-value and low-value jobs. By default, high-value jobs should be given higher execution priority than low-value jobs. Similar to the SLA Load-aware policy, the SLA Value-aware policy evaluates each accepted contract each time a new job arrives in the system. When a new job arrives in the system, the remaining monetary values of the contract are computed against the job. If the remaining values of the contract are sufficiently high, the system issues a rental request to rent additional nodes immediately in order to accommodate the job and future jobs that belong to the contract. In essence, a job that belongs to a contract with high remaining monetary values will be given higher priority for execution. The strategy is to rent additional nodes if the system anticipates that there is potentially a large revenue stream which may be generated by serving jobs from high-value contracts.

More formally, the remaining monetary values of contract sla for job j are computed as follows:

$$remv_{sla} = remv_{sla} - V_j \tag{7.3}$$

where $remv_{sla}$ is the remaining monetary values to be earned from contract sla. The remaining monetary values $remv_{sla}$ are computed each time a job j has been selected to be scheduled. From the $remv_{sla}$, we can determine the maximum revenue for each queued job j as follows:

$$maxoffer_j = (\frac{remv_{sla}}{remd_{sla}}) \tag{7.4}$$

where $remd_{sla}$is the remaining contract duration for contract sla. Instead of prioritizing the queue according to a FCFS (First-Come-First-Served) scheduling scheme (as in the previous SLA Load-aware policy), the scheduler prioritizes queued jobs in descending $maxoffer$ for scheduling. If the $maxoffer_j$ of job j cannot be accommodated due to insufficient nodes, it may rent k_j additional nodes for job j according to the following rule:

$$k_j = \begin{cases} F - S_j & remv_{sla} > S_j(rt_d * uc) \\ 0 & otherwise \end{cases} \tag{7.5}$$

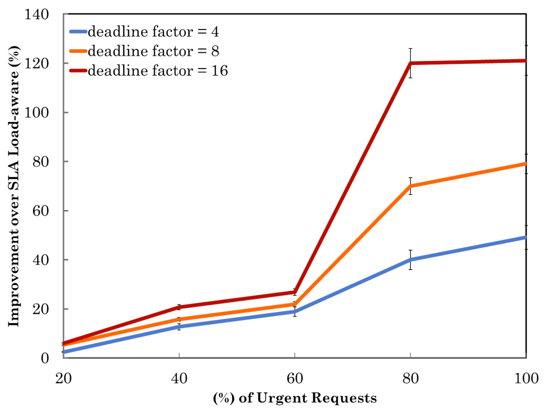
According to the rule above, the system rents k_j additional nodes only if the total rental cost for renting k_j does not exceed the $remv_{sla}$. The rt_d is the minimum rental duration for each rented node, S_j is the amount of requested nodes, F is the amount of currently free nodes, and uc is the rental fee paid per hour for renting a node.

Figure 7.8 shows the profit improvement in percentage terms of the SLA Value-aware scheme compared to the previous SLA Load-aware policy. From Figure 7.8a, we can observe that the improvement in profit increases as the deadline factor increases. In particular, it can be seen that the profit increases exponentially when the percentage of urgent requests increases up to 60%. We can see that improvement of the SLA Value-aware policy substantially increases when the percentage of urgent requests is above 60-70%. This highlights the significant impact of ratio of urgent and non-urgent job requests in the workload.

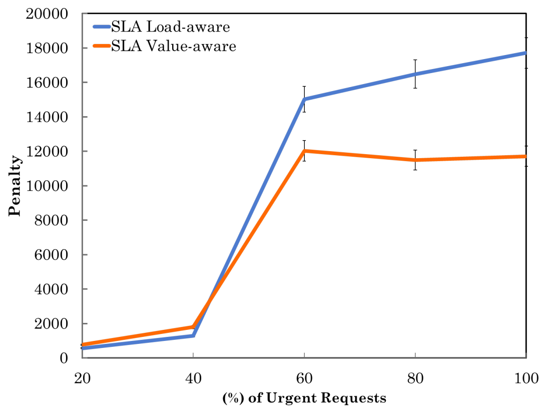
It can also be observed that, when compared to the SLA Load-aware scheme, the SLA Value-aware policy earns additional revenues of up to 20% for a low deadline factor of 4. When the deadline factor increases to 16, the improvement increases up to 120%. This is an encouraging result suggesting that the policy is able to optimize renting decisions very well to accommodate sudden arrivals of incoming urgent job requests without high rental cost. The success is attributed to the effectiveness of the policy in not reacting over-excessively to the sudden arrivals of jobs – only jobs that belong to contracts with high remaining monetary values get special treatment.

Interestingly enough, Figure 7.8b shows that the penalty cost is fairly high for the SLA Value-aware policy when compared to the SLA Load-aware policy at a higher fraction of urgent requests. The result could possibly be attributed to resource competition; low-value jobs are consistently being delayed by the high fraction of urgent jobs. This has a negative effect of violating deadlines for some less valued urgent jobs.

From these observations, it can be seen that there is a fundamental trade-off between the SLA Load-aware and SLA Value-aware policies. We can resolve this by employing a strategy that gives an option



(a) Improvement of SLA Value-aware heuristic.



(b) Penalty cost.

Figure 7.8: Cost Comparison of SLA Load-aware policy versus SLA Value-aware schemes for varying deadline factors.

for the system to switch the priority between high-value and low-value jobs depending on the situation. For example, if a job's deadline is about to expire, the system should execute the job immediately even if it generates less revenue. Moreover, in a situation when it is crucial for the system to ensure no deadline violation for all queued jobs, it may postpone scheduling jobs that belong to the high remaining monetary values contract when it encounters 'riskier' less valued jobs (jobs deadlines which are *nearly* violated). This prompts the development of a more sophisticated cost-aware strategy that takes into account both the jobs' QoS requirements and the system's profit when making scheduling and renting decisions.

7.5.4 SLA Profit-aware

In the previous subsection, the shortcomings of the SLA Value-aware policy lead us to the conclusion that there is a need for a slight modification to the policy. A desirable rental scheme would inherently take into account job deadline and job monetary value as well as the current system's revenue. In this section, we propose a SLA Profit-aware policy that does so.

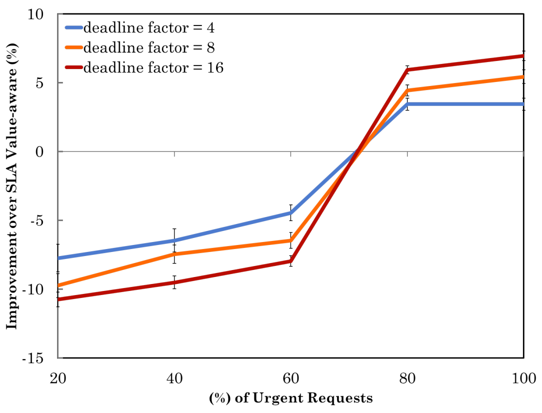
The SLA Profit-aware policy priorities riskier jobs which need to be served immediately before their deadlines are about to expire. If there is no such riskier jobs, the queued jobs will be served in descending order of their monetary values. If a risky job cannot be accommodated with current available nodes and if waiting for the next periodic scheduling interval is likely to violate the job's deadline, the system rents k_j nodes for job j as long as the current profit level plus the remaining monetary values $remv_{sla}$ of the contract *sla* does not exceed the cost of renting $S_j - F$ nodes:

$$k_j = \begin{cases} S_j - F & if(profit + remv_{sla}) \geq S_j(rt_d) * uc) \\ 0 & otherwise \end{cases} \tag{7.6}$$

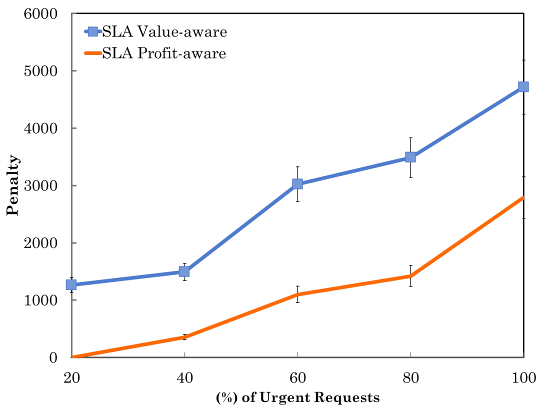
Figure 7.9a shows the profit improvement of the SLA Profit-aware scheme compared to the previous SLA Value-aware policy. As observed, under a low fraction of urgent requests, the SLA Profit-aware policy generates less profit (performs worst) compared to the SLA Value-aware policy. We can see that the profits fall between 5 and 11 percent. However, as the fraction of urgent requests increases, the performance of the SLA Profit-aware policy starts to increase substantially and we can observe that SLA Profit-aware scheme outperforms the SLA Value-aware policy.

The SLA Value-aware policy performs well under a low fraction of urgent requests because there are only a few urgent jobs in the system and there are sufficient nodes to satisfy all the incoming jobs within their QoS requirements. However, as the fraction of urgent requests increases, the SLA Value-aware policy is no longer able to compete with the SLA Profit-aware policy. We can see that at a high fraction of urgent requests, the SLA Profit-aware policy performs better under a high deadline factor. For example, we can see that the SLA Profit-aware policy improves up to 7% under a high deadline factor (deadline factor = 8) over the SLA Value-aware policy.

Nevertheless, it is interesting to see from Figure 7.8b that the SLA Profit-aware policy is able to reduce the penalty cost to zero under a low fraction of urgent requests, whereas the SLA Value-aware policy is unable to keep its penalty cost lower than 1000 even at a low fraction of urgent requests. The positive result of the SLA Profit-aware policy is attributed to its strategy of investing in extra nodes to



(a) Profit improvement.



(b) Penalty cost.

Figure 7.9: Comparison of SLA Value-aware policy versus SLA Profit-aware policy for various deadline factors.

Algorithms	Fraction of Urgent Jobs	
	Small (20%)	Large (80%)
Greedy	16,075	8,226
SLA Rigid	15,454	64,133
SLA Load-aware	16,450	84,162
SLA Value-aware	17,272	117,826
SLA Profit-aware	15,890	121,361

Figure 7.10: Comparisons of Non-SLA-aware and SLA-aware policies by Profit.

accommodate riskier jobs. Although it may not perform as well when compared to the SLA Value-aware policy at a low fraction of urgent requests, it is still able to generate high profits at higher fractions of urgent requests for a different range of deadline factors. It is therefore more effective than the SLA Value-aware policy when it is necessary for the system to guarantee zero deadline violations for applications.

7.5.5 Summary

Table 7.10 summarizes the difference in profits between all of the proposed non-SLA-aware and SLA-aware policies under both a small fraction and a large fraction of urgent requests. We summarize our results as follows:

- All SLA-aware policies outperform the non SLA Greedy policy. Even with a very basic SLA Rigid policy, a slight increase in profit of 2.51% is still observed under workloads with small fraction of urgent requests. The improvement is even higher at 11.8% increase in profit under workloads of a large fraction of urgent requests.
- The SLA Value-aware policy gives the best profit (with a slight 10.7% increase in profit over the SLA Profit-aware policy) for workloads with a small fraction of urgent requests. This demonstrates that the SLA Value-aware policy is the best policy when the SLA consists of many non-urgent job requests.
- The SLA Profit-aware policy gives the best profit (with a slight 8.9% increase in profit over the SLA Value-aware policy) for workloads with a large fraction of urgent requests. This shows that, overall, the SLA Profit- aware policy is the most suitable when the SLA contract consists of a large fraction of urgent job requests over non-urgent job requests.

We make the following conclusion: (i) both SLA Value-aware and SLA Profit-aware policies are very effective as they significantly outperform all other policies across a wide range of conditions and parameters and (ii) the idea of investing and recycling system profit at runtime (as employed by the Profit-aware policy) to accommodate future jobs can actually mitigate the risks of urgent jobs from being dissatisfied and it is a powerful technique for ensuring zero penalty cost in extreme situations of sudden burst of demand for resources.

7.6 Rental Cost Options

In the previous section, several SLA-aware policies have been proposed to guide the resource selection decisions in an attempt to balance the cost of acquiring computational resources and the cost of satisfying application quality-of-service (QoS) requirements. Such SLA-aware policies provide option to rent resources based on short-term and/or long-term planning and to manage these resources according to the needs of its users. However, there is another side to the problem that seems largely untouched. This relates to the symbiosis between users and resource provider: users and resource providers do not operate in a vacuum. There is a clear case for collaborative relationships between them.

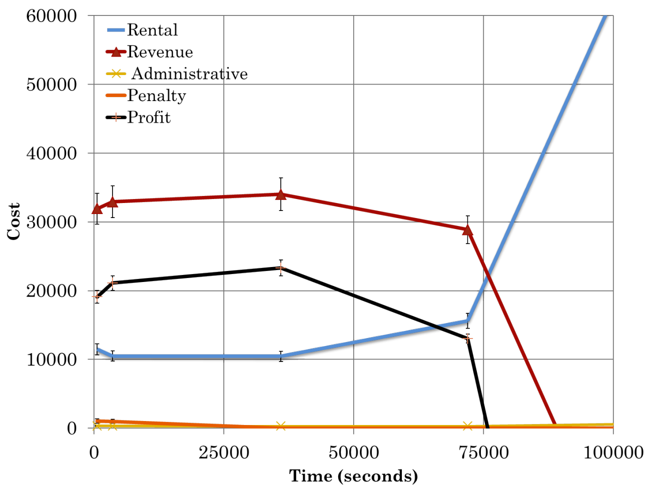
In this section, we answer the following question: *"What is the impact of various SLA-aware policies under different cost options? In particular, are some business models better in the sense of allowing lower rental costs (and therefore cheaper services) for long-term rentals compared to short-term rentals?"* In the context of a global distributed environment, a resource provider may be reluctant to rent out resource nodes for short durations. There are many reasons for this: to promote long-term commitment by users to resource usage, to avoid lower resource utilization, to reduce unnecessary negotiations (communication overhead) with potentially multiple VAs around the world, and to lower administrative costs that can be incurred due to cost overheads from resource deployment. Resource providers may therefore prefer to rent out their resources for long rental durations and they may do so by charging lower rental fees for long rental durations.

The system may offer an option to allow the user to rent on a short-term or long-term basis, or both. Renting on a short-term basis is cheaper for the VA because it can get immediate access to resources when needed without tying up funds in unnecessary rental cost. However, short-term rental is considered expensive from the resource provider's perspective because of the needs to constantly finds its customers. Hence, renting on a long-term basis is cheaper for the resource provider, but it may unnecessarily tie up and thus increase the rental cost for the VA. To examine these various issues, we examine the effectiveness of various SLA-aware policies under different rental and cost conditions in this section.

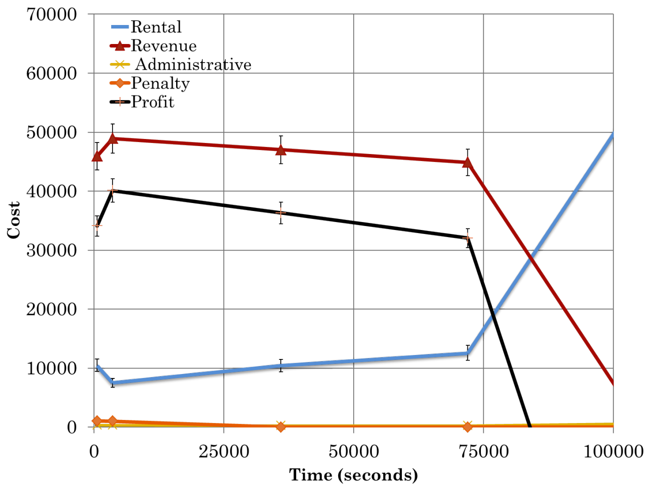
7.6.1 Impact Rental Duration

We first evaluate the impact of renting under increasing rental duration, where each rental duration is fixed for each experiment. We conducted several experiments under various durations: 10 minutes (600 seconds), 1 hour (3,600 seconds), 10 hours (36,000 seconds), 20 hours (72,000 seconds), and 50 hours (180,000 seconds).

Figure 7.11 plots the results for the SLA Rigid policy under both the LLNL Thunder and LLNL Atlas workloads, in which each experiment is repeated for various rental durations. As the rental duration is from 10 minutes to 10 hours, our initial observation is the policy earns higher profit as the rental duration increases. Interestingly, however, as the rental duration continues to increase beyond 20 hours (36,000 secs), the policy no longer delivers higher profit. As observed, the profit falls when the rental duration is over 20 hours. Looking more closely at Figure 7.11, we can see that the rental cost substantially increases as it goes beyond 70,000 seconds (approximately 10 hours). This causes a significant reduction in profit due to a significant increase in rental cost when the rental duration goes above 20 hours.



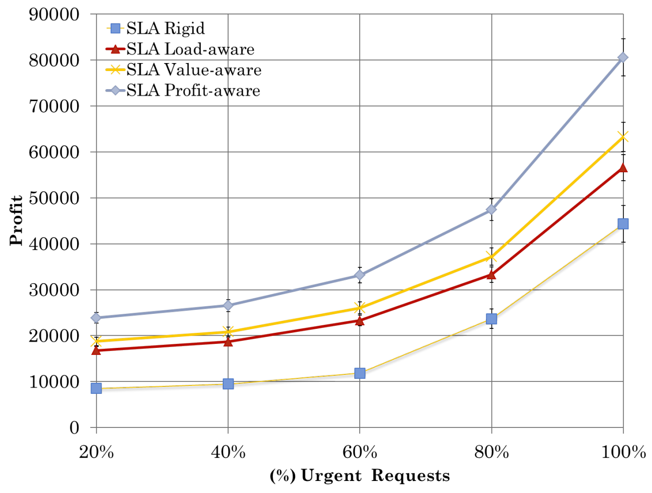
(a) LLNL Thunder at Fixed Duration.



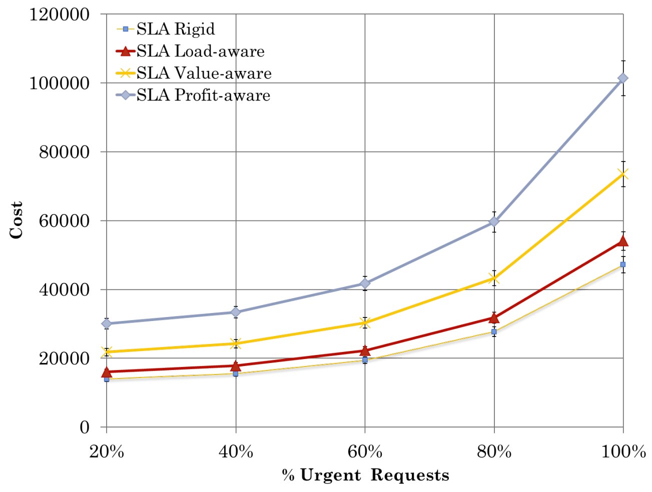
(b) LLNL Atlas at Fixed Duration.

Figure 7.11: Cost breakdown for SLA Rigid policy under increasing rental durations.

At the initial stage, the trend in our results is that the long-term rental option offers a better profit when compared to short-term rentals. However, it reaches a point where long rental duration no longer delivers a higher profit. In our case, for both LLNL Thunder and LNNL Atlas workloads, the optimal rental duration is found to be approximately 10 hours. After this point, the profit starts to degrade substantially.



(a) LLNL Thunder.



(b) LLNL Atlas.

Figure 7.12: Comparison of all SLA-aware policies for both the LLNL Thunder and LLNL Atlas workload traces.

Figure 7.12 shows a comparison of the profits of different SLA-aware policies (i.e., SLA Rigid, SLA Load-aware, SLA Value-aware, and SLA Profit-aware) for the LLNL Thunder workload and for the LLNL Atlas workload as well under an increasing fraction of urgent jobs. When comparing the policies, we can see that the SLA Rigid and SLA Load-aware policies earn lower profits compared to the SLA Value-aware and SLA Profit-aware policies. This is due to the SLA Rigid and SLA Load-aware policies not reacting quickly enough to accommodate sudden changes in workload demands, whilst SLA

Value-aware and SLA Profit-aware policies have better profits especially at a higher fraction of urgent requests because they take into account of execution and rental cost when performing renting decisions.

7.6.2 Impact of a Combination of Short-Term and Long-Term Rentals

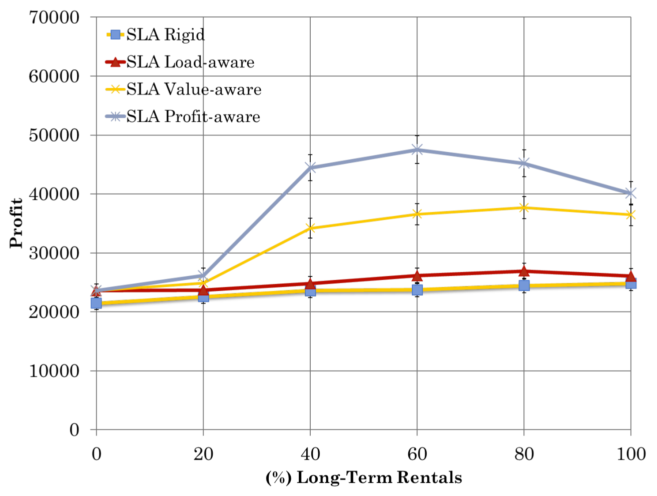
As seen earlier, the cost-aware SLA Value-aware and SLA Profit-aware policies deliver better profit compared to SLA Rigid and SLA-Load aware policies. In the next experiment, we take into consideration the renting of nodes based on a combination of short and long rental durations. In such a case, the system can choose to rent nodes on a short-term or on a long-term basis. Hence, we introduce the possibility for a mixed arrangement of short-term and long-term rentals. The aim is to determine whether there is any cost-benefit in having a combination of short-term and long-term rentals when making rental decisions over the long term to sustain overall profit.

We first evaluate the combination of short-term rentals and long-term rentals, where the cost of short-term rental (short rental) is two times (2x) higher than that the cost of the long-term rental (long rental) option. For the purposes of this experiment, a long rental duration is charged at a standard price of 0.26 per hour, and the charge for a short rental duration is 0.52 (two times higher than long rentals). Furthermore, the short-term rental duration is set to be 10 minutes, whereas the long rental duration is set to be 10 hours. The 10 hours duration is chosen as it was found to be the optimal range, as observed from our previous experiment.

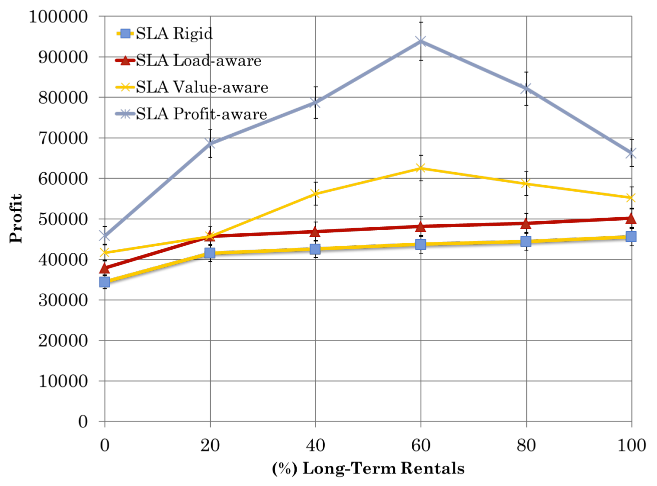
The choice of short-term and long-term rentals is determined accordingly for each experiment. Initially, long-term rentals are initiated prior to the first job execution. Subsequently, the short-term rentals are initiated upon a new job arrival or when a specific threshold condition has been triggered. In such a situation, the number of nodes to rent on a short-term rental basis is determined according to the number of queued jobs, as described in Section 7.5.

The SLA Rigid policy is based on 100% long-term rental with no short-term rental option provided. On the other hand, the SLA Load-aware, SLA Value-aware, and SLA Profit-aware policies have the flexibility to rent nodes based on a combination of both short- and long-rental options. Initially, X% of estimated total nodes (specified from the contract) are rented on a long-term basis prior to job execution. Subsequently, the remaining 100-X% nodes are rented on a short-term basis at different points of application execution to accommodate sudden spikes in demand.

Figure 7.13 plots the results as we vary the mix of long-term and short-term rentals for LLNL Thunder and LLNL Atlas workloads. Initially, all policies earn comparable profits when the percentage of long-term rentals is between 1% and 20%. However, as the percentage of long-term rentals increases beyond this point, we start to see a wide difference in profits between all the different policies. It can be observed that both SLA Value-aware and SLA Profit-aware policies still continue to deliver higher profits beyond 20% of long-term rentals, whereas the profits for the SLA Rigid and SLA Load-aware policies change very little as the percentage of long-term rentals increases. Interestingly, we can also observe that the profits for the SLA Value-aware and SLA Profit-aware policies fall substantially when the percentage of long-term rentals goes above 60%. This shows that too high a percentage of long-term rentals can result in poor performance although the SLA Rigid and SLA Load-aware policies are not



(a) LLNL Thunder.



(b) LLNL Atlas.

Figure 7.13: A combination of both short-term and long-term rental options.

affected much by the percentage of long-term rentals.

We can see that there is not much variation in profit across increasing percentage of long rentals for both SLA Rigid and SLA Load-aware policies. This may be attributed to the additional cost overhead incurred from nodes being tied up unnecessarily as well as the penalty cost incurred from the burst-arrival of new jobs. On the other hand, both the SLA Value-aware and SLA Profit-aware polices are able to continue delivering significantly higher profits as the percentage of long-term rentals increases. This is attributed to the cost-aware approach employed by both policies to rent only an optimal number of nodes when reacting to changes in demand. We observe a similar trend for the LLNL Atlas workload as well. In fact, we can observe that both the SLA Value-aware SLA Profit-aware policies earn significantly higher profits compared to the other policies at all percentages of long-term rentals. This shows that they both perform very well under small or medium short duration jobs and for many parallel jobs that run for longer duration.

Figure 7.14 further shows a cost breakdown for the SLA Profit-aware policy under a fixed rental

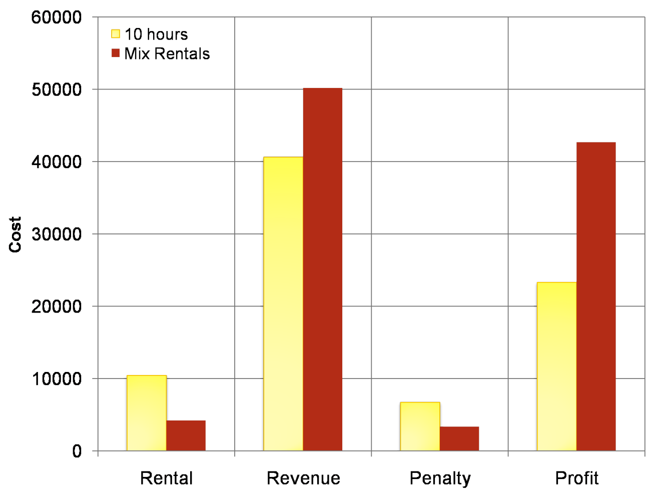


Figure 7.14: Fixed rentals vs. mixed rental arrangements for the SLA Profit-aware policy.

duration, where all rented nodes are rented on fixed rental durations each with 10 hours versus a mix of rental arrangements at 50% long-term rentals. We can observe that a higher profit can be achieved under a mixed rental arrangement compared to a fixed rental duration. Overall, these results suggest that renting a combination of short-term and long-term rentals is more effective in comparison to a rental arrangement policy where each node is rented on a fixed duration basis.

7.6.3 Impact of Short-term and Long-term Rental Options under Varying Rental Durations

So far, we have examined a mixed rental arrangement of short-term and long-term options. Previously, a short-term rental was defined a rental duration of 10 minutes and a long-term rental option has a rental duration of 10 hours. *What would be the impact of having different durations for each short-term and long-term rental option?*

Next, we investigate the impact of profit for a combination of short-term and long term-rentals under different rental durations. Figure 7.15 shows the comparison of profits for the SLA Profit-aware policy under the following rental durations: (1) short-term rental of 10 minutes and long-term rental of 1 hour [10 mins, 1 hour]; (2) short-term rental of 10 minutes and long-term rental duration of 10 hours [10 mins, 10 hours]; (3) short-term rental of 1 hour and long-term rental of 10 hours [1 hour, 10 hours] and finally (4) short-term rental of 1 hour and long-term rental of 50 hours [1 hour, 50 hours]. We restrict our discussion here to the SLA Profit-aware policy because we have already observed earlier that all SLA-aware policies follow similar trends and the SLA Profit-aware policy is the best performer. For the same reason, we also restrict our experiments to the LLNL Thunder workload traces only.

Figure 7.15 plots the results under an increasing percentage of long-term rentals. As can be observed, the [1 hour, 10 hours] rental arrangement delivers the highest profit compared to the rest of rental options. In contrast, the [10 mins, 10 hours] and [1 hour, 50 hours] rental options earn the worst profit. These results show that the rental duration assigned for short-term and long-term rentals can have different outcomes in relation to profit. From this observation, we can conclude that a combination of too

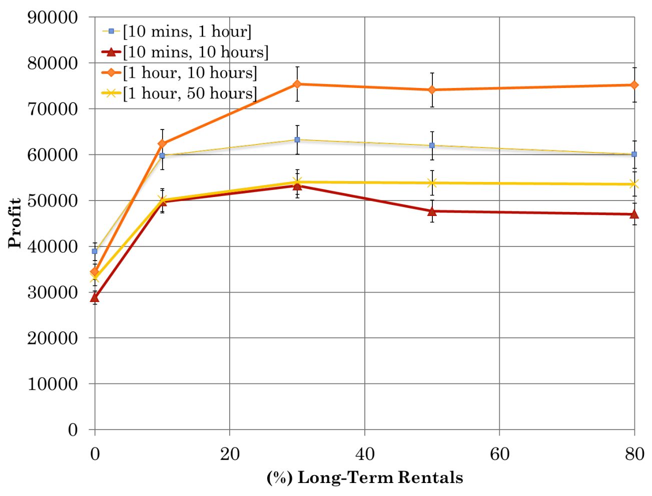


Figure 7.15: Varying durations for short-term and long-term rentals.

short a duration for the short-term rental option and too long a duration for the long-term option can have a detrimental effect on the overall profit. For example, we can see that the short-term rental option should not be too short (10 minutes) and the rental duration for long-term rental option should not be too long (50 hours). In our case (as observed for the LLNL Thunder workload), the highest profit is obtained when the short-term rental duration is set to 1 hour and the long-term rental duration is set to 10 hours.

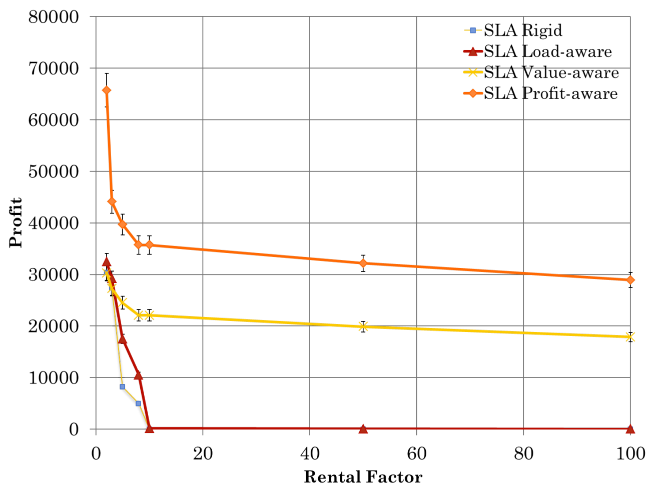
7.6.4 Impact of Rental Cost

In the previous experiments, we assume that the cost for short-term rental is two times (2x) higher than that of the long-term rental. The question is: *what would happen if the cost for short-term rentals increases at a rate greater than 2x?*

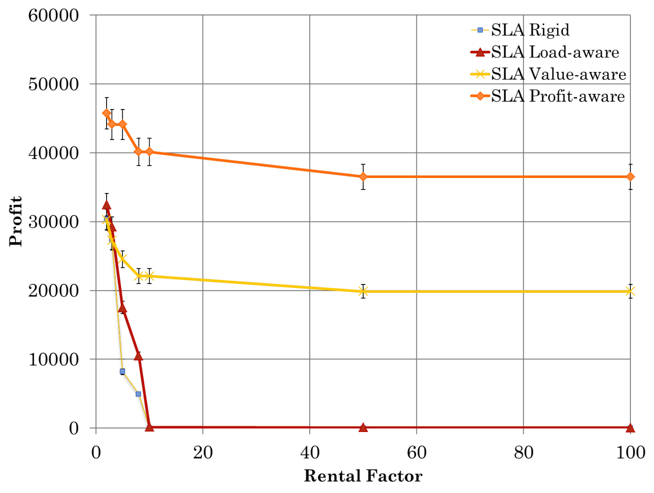
In practice, a resource provider is more likely to charge a very high rental fee for short-term rentals compared to long-term rentals. There are many reasons for this: to promote long-term commitment to renting resources, to ensure an increase in resource utilization, and to lower administrative costs that can be incurred in the negotiation and deployment phases when there are potentially multiple VAs competing for resources.

Figure 7.16 plots the impact of profits for different rental policies under increasing rental factor. In the simulations, increasing the rental factor to x value leads to increasing the cost of renting a resource node by x times. For example, a factor of 4x implies that the cost to rent a resource nodes is 4 times higher. Generally, large rental factor leads to an increase in rental cost for short-term rental which results in lower profit. Therefore, the policy must aim to reduce the number of short-term rentals by accurately renting acceptable nodes to accomodate new arrival jobs (burst) in near future.

As can be seen, the SLA Rigid policy's profit drops off rapidly as the rental factor increases and its profit drops nearly to zero when the rental factor is greater than 10. This is attributed to the policy's static approach in renting exactly N nodes upon contract arrival. Therefore, there is no room for improvement in renting additional nodes since this would incur significantly high rental cost due to idle nodes. The SLA Load-aware policy also delivers similar poor performance, although it offers slightly higher profit



(a) Increasing rental factor with static rental duration.



(b) Increasing rental factor with 50% long-term rentals.

Figure 7.16: Impact on profit of increasing rental cost.

than the SLA Rigid policy. In contrast, the increase in rental factor does not stop either the SLA Value-aware or SLA Profit-aware policy from delivering a high profit, although we can see that the profit falls slowly with increasing rental cost. For example, when the rental cost factor increases beyond 10, we can see a reduction trend in profits. We can also observe that the SLA Profit-aware policy earns substantially higher profit than the SLA Value-aware policy. The SLA Profit-aware policy earns more profit because it takes into account the cost of renting additional nodes, whereas the SLA Value-aware policy only takes into account job monetary values from queued jobs.

Figure 7.16b shows the same experiment repeated for 50% long-term rentals. We can see that similar trends are observed. However, for both SLA Value-aware and SLA Profit-aware policies, we can observe that the profits do not fall significantly as the rental cost factor increases. This is in contrast with our previous experiment that was carried out with static rental durations. This result indicates that both SLA Value-aware and SLA Profit-aware policies are less sensitive under an increasing rental cost factor when a combination of mixed rental durations is used.

7.6.5 Impact of Monetary Values

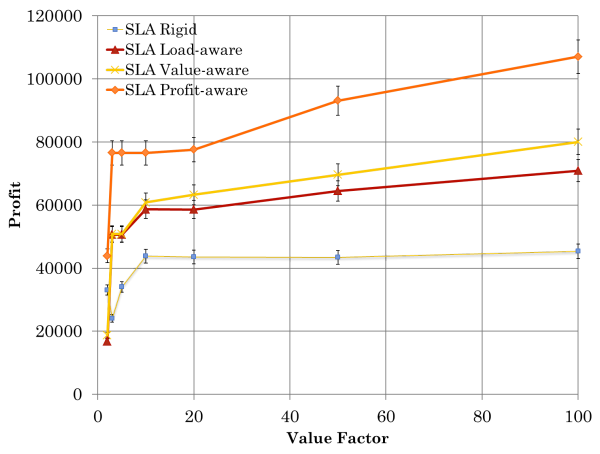
We now examine profit sensitivity to variations in monetary values to examine their impact across all SLA-aware policies. We aim to determine whether there are any benefits in making use of monetary values information specifically for the cost-aware approaches: SLA Value-aware and SLA Profit-aware policies that take into account monetary values information when performing renting decisions. In the simulations, increasing the value factor to x value leads to increasing the monetary value by x times. For example, a factor of 4x implies that the monetary value assigned is 4 times higher.

Figure 7.17a plots the profits across all SLA-aware policies with static rental duration under an increasing value factor. We can observe several interesting trends from the results. For static rental duraction, it can be observed that the SLA Profit-aware policy substantially outperforms all other policies by delivering a significant increase in profit as the value factor increases. As expected, the SLA Value-aware policy also delivers slightly higher profit compared to the SLA Load-aware and SLA Rigid policies. Figure 7.17b further shows the profits for all SLA-aware policies with 50% long-term rentals as a function of increasing value factor. Again, we see a similar trend, but with the exception that the increase in profit for all SLA-aware policies is much more significant. In particular, we can see a linear increase for the SLA Profit-aware as the value factor increases.

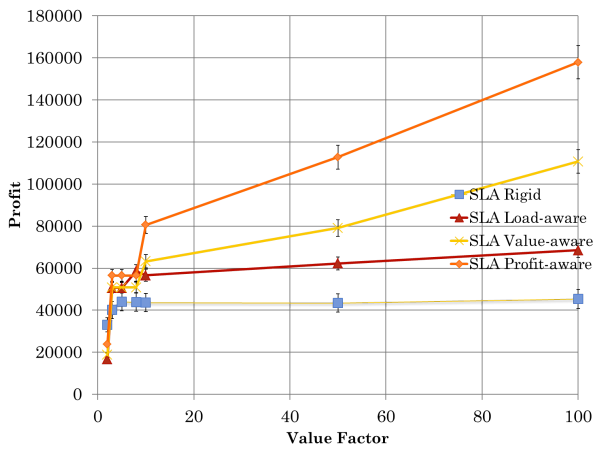
7.6.6 Impact of Sudden Spikes in Demand

A highly variable and chaotic workload occurs because users often submit jobs in bursts as opposed to submitting individual jobs separated by shorter periods of idle time. *How efficient are the various SLA-aware policies in adapting to sudden spikes in demand?*

We therefore aim to examine the performance sensitivity to burstiness in the workload. In order to this, we make some changes to our workload model in order to examine the sensitivity of the various SLA-aware policies under different levels of burstiness as a function of job burst sizes. To simulate the job burst sizes, the jobs from the LLNL Thunder workload traces are grouped into a number of sets. Each set is assigned a job burst size where a fixed number j of jobs is assigned per burst. The greater the



(a) Increasing value factor with static rental duration.



(b) Increasing value factor with 50% long-term rentals.

Figure 7.17: Profit sensitivity to increasing monetary value.

burst size, the higher the number j of jobs are submitted. At any one job submission, a fixed j of jobs is submitted to the system. Since this will have some effect on utilization, the job interarrival time is fixed with an exponentially distributed service time with a mean of 0.15 hours in order to highlight relative comparisons of the different SLA-aware policies.

Figure 7.18 plots the results for all the SLA policies under increasing burst size. For all policies, the general trend is that as the burst size increases, all policies suffer some profit degradation due to cases where bursts include multiple high-value jobs (i.e., a large number of urgent jobs) which compete directly against each other. We can see that as the workload becomes chaotic (burst size increases), the profits for both the SLA Rigid and SLA Load-aware policies suffer rapidly as burst size increases. In particular, the profit for the SLA Rigid policy becomes almost zero when the burst size approaches 8. In contrast, the SLA Value-aware and SLA Profit-aware policies demonstrate significantly less degradation in profit with larger burst sizes. However, both the SLA Value-aware and SLA Profit-aware policies still suffer from profit reduction at high burst sizes. Nonetheless, the SLA Profit-aware policy is able to greatly minimize profit degradation due to its cost-aware approach because it is able to determine the optimal number of nodes to rent based on the information provided from the remaining total monetary

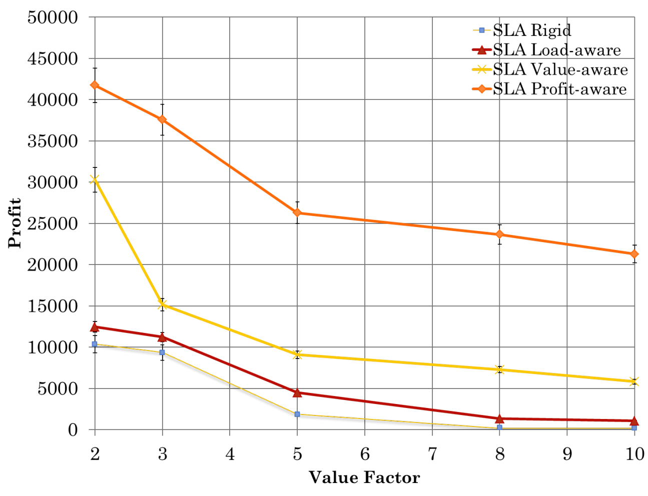


Figure 7.18: Impact of burst size.

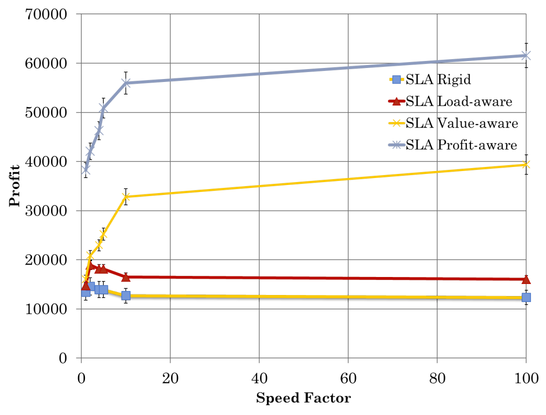
values (TMV) of the contract in addition to the monetary value information provided by new job arrivals.

7.6.7 Impact of Node Heterogeneity

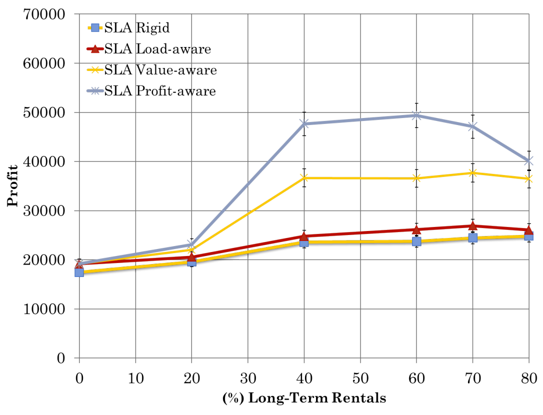
In the previous experiment, we can see that, in all cases, overall profit is superior when applying the SLA Value-aware and SLA Profit-aware policies. In most cases, the SLA Profit-aware policy delivers higher profits compared to the rest of the policies. The question now is: *"would we have similar observations under rentals of heterogeneous nodes with different processing speed capacity?"*

We therefore aim to examine the effect of the variation of node capacity under an increasing percentage of long-term rentals. A node consists of two parameters: processing speed and speed factor. Here, the completion time of a task j on node n is calculated based on the node's processing speed. Therefore, the speed factor β is used to assess the capacity of each node against a standard node. For instance, a node with 2x speed factor would incur two times extra rental fee cost compared to a node with standard processing power. We carry out simulations for various capacity nodes under a symmetric case (almost all processors follow the same capacity) and an asymmetric case (high deviation in the processor capacity).

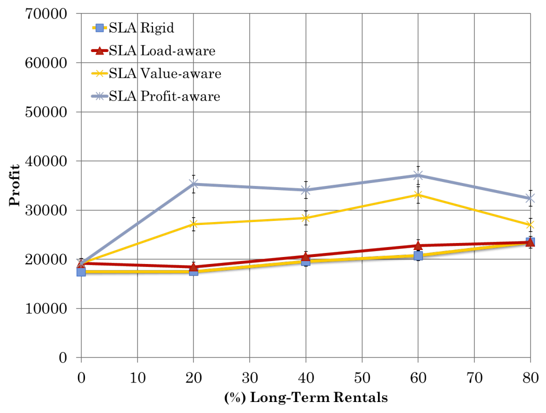
Figure 7.19a shows the profits for all SLA-aware policies a symmetric case where we assume that the capacity of each node follows a normal distribution with a standard deviation of 1 percent of the respective mean capacity value β where β is defined to as ($\beta = 10$). We can see two different trends. Initially, for both the SLA Rigid and SLA Load-aware policies, we can see an immediate reduction in profit as the speed factor increases. We attribute the reduction in profit to the high rental cost incurred in renting fast nodes. However, both the SLA Value-aware and SLA Profit-Aware policies display the opposite trend: we can observe that the profits for both the SLA Value-aware and SLA Profit-aware policies increase at a significant rate until the speed factor goes above 10 when the rate of increase starts to slow down. This shows that the cost-aware approach is very effective in optimally renting the amount of resource nodes while lowering the penalty cost by accommodating the job QoS requirements. As a result, high profit can be obtained even under increasing processing capacity. On the other hand, the SLA



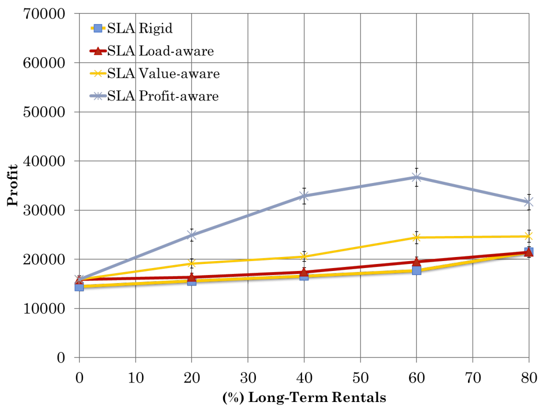
(a) Increasing Speed Factor β.



(b) Increasing long-term rentals for $\beta = 10$.



(c) asymmetric biased low case (20% high capacity nodes).



(d) asymmetric biased high case (80% high capacity nodes).

Figure 7.19: The impact of node heterogeneity.

Rigid and SLA Load-aware policies are unable to maintain their profits with increasing mean processing capacity.

Figure 7.19b further shows the effect of increasing long-term rentals under a high-capacity node condition, where the mean processing speed factor is set to 10 ($\beta = 10$) with a high standard deviation of 20 percent of the respective mean speed factor value. As expected, we can observe that both the SLA Value-aware and SLA Profit-aware policies are able to maintain a high profit as the percentage of long-term rentals increases up to its optimal level which is 60 percent. On the other hand, the SLA Rigid and SLA Load-aware policies struggle to earn high profits as the percentage of long-term rentals increases.

Next, we consider the asymmetric case where each node is either assigned a low-capacity or a high-capacity node. Each node follows a normal distribution in terms of processor capacity with a high standard deviation of 20 percent of the respective mean speed factor value. The mean speed factor of the low-capacity node is set to $\beta = 1$, while the mean speed factor of the high-capacity node is set to $\beta = 10$. Figure 7.19c presents an asymmetric-biased low-capacity case where the majority of the nodes are of low capacity (80% slow and 20% fast nodes), while while Figure 7.19d presents an asymmetric-biased high-capacity case where the majority of the nodes are of high capacity (80% fast and 20% slow nodes).

For these two cases, we can observe that the existence of the different node groups with extreme processing capacity leads to significant deterioration of profit for both the SLA Rigid and SLA Load-aware policies. In contrast, both the SLA Value-aware and SLA Profit-aware policies are able to earn high profits for both the asymmetric-biased low case and the asymmetric-biased high case. This is due to the cost-aware approach adopted by both the SLA Value-aware and the SLA Profit-aware policies which means that they are able to optimize the rental decision based on the monetary values information provided about the SLA contract and individual job monetary value.

Interestingly, for these two cases, we can observe that the SLA Value-aware policy does not perform very well under the asymmetric-biased high case compared to under the asymmetric-biased low case. We can see a huge difference in profit between the asymmetric-biased low case and asymmetric-biased high case, although it is still comparably better than the profit of the SLA Rigid and SLA Load-aware policies. On the other hand, the SLA Profit-aware policy delivers high profits for both asymmetric cases.

7.7 Summary

We have presented a SLAM framework that is a scalable and dynamic approach for creating and managing SLAs. Based on this framework, we performed an evaluation of various SLA-aware policies and presented a comparative analysis of their performance under a wide range of cost options imposed by resource providers. Our results provide interesting key insights into the applicability and effectiveness of the SLA-aware approaches in the VA environment:

- The combination of short-term and long-term rentals delivers higher profit compared to rental arrangements that are based on static and fixed rental durations.
- Higher allocated monetary values do not have significant effects on the SLA Rigid and SLA Load-

aware policies under the mixed rental arrangement consisting of short-term and long-term rentals. However, the profits for both the SLA Value-aware and SLA Profit-aware policies are significantly higher compared to those of the other policies under a mixed rental arrangement of between 40% and 80% long-term rentals.

- Increasing the rental cost has the effect of decreasing profitability for all policies. However, both the SLA Rigid and SLA Load-aware policies suffer significantly when the rental cost factor goes above 10; in which their profits turn to losses. Both the SLA Value-aware and SLA Profit-aware policies are able to adapt to higher rental cost scenarios because they are able to optimize the number of rented nodes from the information derived from the SLA contract.
- The SLA Value-aware policy does not operate well under asymmetric node heterogeneity where there is a high deviation between high- and low-capacity nodes. On the other hand, the SLA Profit- aware policy is able to adapt to both asymmetric-biased low and asymmetric-biased high cases because it takes into account current earning profit when determining the number of nodes to rent. This shows that the SLA Profit-aware policy is more efficient when compared to the other SLA-aware policies, especially when the workloads are bursty and when there is a large heterogeneity and high deviation in the node capacity.

The above insights and findings indicate that SLA cost-aware approaches are very encouraging for the scheduling of applications and jobs which have highly variable and chaotic workloads. In particular, both the SLA Value-aware and SLA Profit-aware policies are very promising since they are adaptable to a wide range of rental options and parameters. In contrast, the SLA Rigid and SLA Load-aware policies fail to deliver high profits in many instances due to the lack of knowledge and information needed to rent the optimal number of nodes in order to serve jobs' QoS requirements.

7.8 Discussion and Comparison to Related Work

Service level agreements (SLAs) have been proposed to differentiate the importance or priority of jobs since they define service conditions that both the user and the system must agree upon before the application is accepted for execution (Yeo and Buyya, 2005). Much research has been undertaken to identify SLA standards to define and manage such services in a distributed resource environment. Proposed architectures, such as the Open Grid Services Architecture (Foster, (eds.) And U, 2003), have attempted to take advantage of XML-based Web Service standards such as the Simple Object Access Protocol (SOAP) (Chester, 2001) and the Web Service Description Language (WSDL) (Christensen et al., 2001) to facilitate the definition of quality of service (QoS) and service level agreements (SLA) in computational Grid settings.

In recent years, there has been a significant amount of research on various topics related to SLAs. Issues related to their overall incorporation into resource management system architectures have been discussed in (Fakhouri et al., 2001; Keller and Ludwig, 2003; Stuer, Sunderam, and Broeckhove, 2005; Eickermann et al., 2007; Hudert, Ludwig, and Wirtz, 2009). Issues related to the specification of SLAs have been considered in (Keller and Ludwig, 2003; Stuer, Sunderam, and Broeckhove, 2005; Green et al., 2007; Litke et al., 2008). Issues specifically related to and motivated by the usage of SLAs for resource management in distributed systems have been considered in (Liu, Squillante, and Wolf, 2001; Czajkowski et al., 2002; John et al., 2003; Zhang and Ardagna, 2004). Several works have examined issues related to trust and security (Xiong and Perros, 2006; Tiwari, Dwivedi, and Karlapalem, 2007; Clark et al., 2010). Later works have also drawn upon relevant research carried out particularly in the context of agents (Chhetri et al., 2006; Qiang He, 2006; He et al., 2009), where several techniques have been used to model negotiation, ranging from heuristics to game theoretic and argumentation-based approaches (Jennings et al., 2001; Siddiqui, Villazón, and Fahringer, 2006; Rubach and Sobolewski, 2009). More recently, a significant area of research has focused on the economic aspects associated with the usage of SLAs for service provision (e.g., charges for successful service provision, penalties for failure, etc.) (Liu, Squillante, and Wolf, 2001; Barmouta and Buyya, 2002; John et al., 2003; Zhang and Ardagna, 2004; Yeo and Buyya, 2005). However, relatively little research has been undertaken on SLA-based job scheduling.

The lack of support functionality such as SLA management and the difficulty in providing QoS management in a distributed environment means that many of today's resource management systems operate on a best-effort basis. Best-effort systems are characterized by a lack of service level guarantees, and the quality of service they provide may vary over time due to workloads, resource availability, etc. Even today, well-known resource management systems such as Condor (Thain, Tannenbaum, and Livny, 2005), Load Sharing Facility (LSF) (Grimshaw, Wulf, and Team, 1997; Xu, 2001), Portable Batch System (PBS) (Henderson, 1995), and Sun Grid Engine (SGE) (Gentzsch, 2001a) still adopt system-centric approaches that only optimize overall cluster performance such as processor throughput, utilization, average waiting time and response time for jobs. These systems neglect and ignore user-centric required services that truly determine applications' needs and quality of service requirements. There are no or minimal means for users to define QoS requirements such as their job deadlines and their valuations during job submission so that the system can improve cost and performance. Resource reservation has always been the standard approach in securing access to resources in distributed environments. However, this approach is not effective because it comes at too high a cost. For example, if reservations are made but jobs complete their execution earlier, resources cannot be made freely available to other applications (F.Heine et al., 2005). Scheduling a particular reservation also require strict planning immediately prior to application execution and the user must provide information on exactly how long they plan to run their application (Park and Humphrey, 2008), and users must abide by this constraint. All of these situations result in resource utilization that is far less than the overall capacity.

With regards to SLAs that make use of monetary values, there are already several resource management systems that make use of application monetary valuations to steer scheduling decisions. For example, REXEC (Chun and Culler, 2000) is a remote execution environment for a group of workstations that adopts an auction-based resource allocation strategy. The strategy is to assign resources proportionally to each job based on the user's bid (valuation) for that job. Tycoon (Lai et al., 2004) also adopts the same bid-based proportional share technique as REXEC, but extends it with continuous bids for allocating resources in a distributed environment. Our SLAs also make use of monetary values information to guide scheduling and renting decisions, but we do not employ auctions as part of the decision-making process.

Instead, our work focuses on making effective scheduling and renting decisions by employing a cost-aware approach. We tackle the renting issues in the context of a VA that rents resources rather than owns them, and the VA is responsible for managing these resources in the most effective manner to increase the profit of the system by maximizing the system's revenue. In this regard, Popovici and Wilkes (2005) proposed a similar system model to our work in which a VA rents resources instead of owning them. However, the work does not explore the advantages of making use of SLAs to satisfy the job QoS requirement and to minimize rental cost. More recent work includes AuYoung et al. (2006) who propose a higher-level contract similar to our SLA model that captures the overall and the aggregate application QoS requirements. This is perhaps the closest to our work, but the work still does not explore SLA-aware policies that take into account important parameters such as job deadlines, remaining total load, total monetary value, and penalty cost. To the best of our knowledge, our work is the first attempt that investigates SLA-aware policies under a controlled environment where both users and resource providers are operating on various rental cost options and rental arrangements.

7.9 Chapter Summary

In this chapter, we presented a SLA management framework where explicit service level agreements (SLAs) are used to increase the overall application satisfaction while minimizing associated rental cost. Key issues are: (i) how to deal with the case where the job parameters (e.g., job execution time) provided by the users are not accurate and (ii) how to guide renting decisions under demand uncertainty.

We have presented a SLAM framework that is a scalable and dynamic approach for creating and managing SLAs. Based on this framework, we introduced several SLA-aware policies that aim to minimize the cost of renting resources by making use of application QoS (i.e., deadline) requirements and system parameters such as the estimated total load, contract duration, contract TMV, and estimated number of generated jobs, in addition to individual job information.

We performed an evaluation of various SLA-aware policies and presented a comparative analysis of their performance under a wide range of cost options imposed by resource providers. Our results provide interesting insights into the applicability and effectiveness of the SLA-aware approaches under the VA environment. In particular, we found that the SLA Profit-aware policy is very effective, outperforming all other policies across a wide spectrum of rental arrangements and rental options.

Chapter 8

Conclusion

The experience gathered from our practical experience with and observations on the deployment and use of current distributed computing infrastructures for distributed and high performance computing projects over the last few years has enabled us to recognize the difficulties in operating distributed resources on a global scale and reconciling the interests of users, applications and the various resource providers from different administrative organizations.

In large-scale distributed computing environments, there is a large number of available resources and a large base of users who would like to make efficient use of the distributed resources to perform computationally intensive jobs and tasks. Both the users and the resources are generally distributed across the globe and they belong to a large number of institutions and/or resource providers. The nature of the problem we are trying to solve in this thesis is very straightforward: how can we effectively and efficiently utilize such geographically distributed resources while still delivering quality of service for users?

The management of distributed resources in a large-scale distributed environment is one of today's most complex challenges and the problem is very difficult to solve. The current solution is Grid computing which emerged in the late 1990s; it provides the infrastructure for dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities. Grid computing provides a platform for co-ordinated resource sharing and problem solving in multi-institutional virtual organizations (Foster et al., 2002). The meta-scheduling framework is the core principle of the resource sharing concept in current Grid infrastructures. In such a meta-scheduling approach, a single global scheduler collects all job requests and collects information on resource availability and resource utilization information across all resource providers so that computation and communication tasks can be scheduled more efficiently. This thesis has discussed the various meta-scheduling issues in a distributed scheduling environment; particularly the scalability issues encountered with a global meta scheduler, the resource allocation delays and other QoS aspects including the dynamic QoS of running applications.

Scalability of management becomes an issue because Grids become physically distributed and have thousands of concurrent users. A single global scheduler will soon become overloaded and inefficient when trying to allocate distributed resources for multiple simultaneous requests. We have also stressed the difficulty encountered when sharing resources among different providers and the inability to specify user-level QoS fairly due to the lack of cost modelling and lack of SLAs in such environments. The QoS support also becomes an issue because some users have jobs with strict deadlines, while others are less restricted, and some may have different QoS requirements during the execution of a job. A framework for scheduling urgent jobs sooner that does not rely on simply trusting user input is needed. In this thesis, we have proposed a cost-aware scheduler to take into account the actual operating costs of machines, performance trade-offs in operating computational resources to serve jobs, and how to perform jobs in the most cost effective manner.

Our attempts to resolve these issues have led to the contributions presented in this thesis. A virtual authority (VA) has been introduced in the thesis as a collection of resources controlled, but not necessarily owned, by a group of users or an authority representing a group of users. The VA leverages the fact that different resources in disparate locations will have varying usage levels. By creating smaller divisions of resources called 'Virtual Authorities', users would be given the opportunity to choose between a variety of cost models, and each VA could rent resources from resource providers when necessary, or rent out its own resources when underloaded. The resource management is simplified since the user and owner of a resource recognize only the VA because all permissions and charges are associated directly to the VA. Furthermore, the VA establishes a trust relationship with the resource provider through SLA agreements and therefore individual jobs/tasks which are associated with individual users are controlled by the VA and not the resource provider. Effectively, building upon on a multi-tier design and framework, the VA is independent from competitors and can manage its own resources. The VA concept is the cornerstone of our approach in the thesis because it is based on the notion that all participants are driven by their own self-interests when collaborating in the context of a large-scale resource sharing environment.

Although both users and resource providers have their own self-interests and act independently, the collaborative co-ordination between them is very important to ensure full co-operation in delivering on the QoS requirements of users while at the same time taking into account the self-interests of resource providers. There is a need for effective resource provisioning and service level agreements mechanisms (as investigated by the means of effective rental policies and SLA-aware policies in this thesis) in order to optimize the job scheduling process while reducing the resource management overheads. Focusing specifically on distributed and high performance computing (HPC) applications, we have proposed in this thesis a cost-effective framework that aims to improve the overall performance, scalability and cost manageability of a resource management system. In summary, this is achieved by the following innovative approaches:

- *Virtual Authority Framework:* We have introduced the VA, which is based on a multi-tier paradigm that separates the concerns of application management, job scheduling, and resource renting. The VA is composed of an Application Agent (AA), a VA Scheduler, and a VA Negotiator. It is the role of the AA to translate the processing needs of the job into actual specific resources. Because users generally might not be aware of the resource requirements of their job, the AA may depend on rules embedded in the application by developers. It is also the role of the AA to enable transparent execution on remote nodes irrespective of the physical location of the nodes. The VA Scheduler assigns jobs to resources and attempts to observe QoS requirements. The VA Scheduler schedules jobs based on an amount of virtual currency which could be submitted along with a job depending upon how urgent that job is; a limited amount of this currency would prevent users from trying to make all of their jobs seem urgent. Based on the urgency and resources needed for a job, the VA Scheduler attempts to make the most efficient possible use of the resources at hand to complete a job within the necessary time frame. Finally, the VA Negotiator contacts and negotiates with other resource providers to rent out resources when necessary. It is controlled by a 'rental' policy which is supported by a pool of resources that the system may rent from external resource providers. The main objective of the multi-tier VA framework is to offer a scalable and cost-competitive platform that can attract users to run their distributed and parallel high performance computing (HPC) applications. In Chapter 4, we have demonstrated the feasibility of the VA framework through the design and implementation of the HASEX proof-of-concept prototype.

- *Cost Framework and Analysis:* The VA framework relies on the market dynamics of supply and demand (penalty cost vs. rental cost) to provide the optimal QoS for the price users are willing to pay. A cost-benefit analysis of current distributed infrastructures and resource management infrastructures such as a private cluster system, a large dedicated HPC facility and a Grid infrastructure was expected to enable us to gain a better understanding of how to derive these costs. Therefore, in Chapter 5, we performed a comprehensive analysis of the cost-benefit of the LUNAR project (Sumby, 2006), the Lawrence Livermore National Lab (LLNL) (Feitelson and Rudolph, 1995) computing facility, and EGEE Grid (Berlich et al., 2006). From our analysis, we were able to measure the economic viability of the various resource management system set-ups and formulate a novel costing framework that considers a wide range of parameters such as job/task execution deadlines, job monetary values, and resource cost. Our proposed costing model effectively provides a mechanism for the VA to quantitatively evaluate its conflicting objectives of lowering rental-related costs subject to meeting application-level QoS constraints.

- *Rental Policies:* It is helpful to understand how the VA determines the number of nodes to rent and when to rent them. In Chapter 6, we presented a set of rental policies that guides the VA to rent external resources from resource providers. Based on the costing framework we have proposed, a rental policy was designed to provided a set of basic rules to determine the amount of nodes to rent and when to rent them. The rental policies explicitly make use of job execution deadlines and virtual currency (i.e., monetary values) parameters to guide rental decisions. By using real workload traces data from the LLNL computing facility, we evaluated our proposed cost-aware scheduler and rental approaches under varying rental options and system parameters. Our results provide insight into the benefits of cost-aware rental policies for achieving high QoS satisfaction with minimal resource costs to deliver high profits. Our exploration of and evaluation on various rental policies serve as a foundation for improving productivity and return on investment by satisfying demand without a heavy upfront investment and without the cost of maintaining idle resources.

- *Service Level Agreements:* A renting mechanism will only be efficient when there is a trust relationship between the VA and resource providers. SLAs offer an attractive platform for describing all expectations and obligations in the service commitment relationship between the VA and resource providers to promote long-term rentals. In Chapter 7, to promote long-term rentals, we have proposed an SLA management (SLAM) framework with several SLA-aware policies that make use of additional system parameters such as the estimated total load, contract duration, contract TMV (total monetary value), and estimated number of generated jobs. We then performed an evaluation of the SLA-aware policies and presented a comparative analysis of their performance under a wide range of cost options imposed by resource providers. Our results provide interesting insights into the applicability and effectiveness of the SLA-aware approaches under the VA environment.

The primary focus of the thesis has been on the proposal of effective mechanisms that can largely self-manage, and that provide better resource availability, performance and scalability at lower cost. We introduced a VA system that can be constructed from rented resources and proposed a set of rental policies and SLA-aware policies that can make such a system a viable alternative to the current resource-sharing computing paradigm. We have demonstrated through our contributions in this thesis, that our innovative approaches can provide improved QoS for applications while lowering resource cost effectively. However, there are many potential issues that may be associated with the approaches that are not specifically addressed by the contributions of the thesis, such as resource discovery, the market dynamics of supply and demand and security. These domains are worthy of further exploration in future work.

8.1 Contributions of the Thesis

We will now examine in more detail the contributions of this thesis.

8.1.1 Virtual Authority

In Chapter 3, the Virtual Authority (VA) is introduced as a means to provide a cost-effective framework for supporting distributed and parallel high performance computing applications (HPC) with minimal infrastructure and resource costs. The VA concept is based on a multi-tier model that resolves the limitations associated with current distributed computing and resource management by separating the roles of application management, job scheduling, and resource provisioning. The VA framework has three distinct tiers. The upper tier offers the ability for the end users to submit jobs using conventional batch job submission interfaces. Alternatively, adaptive and interactive steering applications may request resources directly using simple API calls. The calls are handled by the application agent (AA) which resides between the application and the middle tier to provide a flexible execution environment where the compute capacity can be adapted to fit the needs of applications as they change during execution. The middle tier makes use of the QoS information provided via the AA and appropriately schedules application jobs based on job requirements. The lower tier consists of a VA Negotiator that issues rental requests for additional resources to external resource providers in accordance with rental policies and SLA agreements.

Based on the VA framework, we presented the design and implementation of the HASEX proof-of-concept prototype in Chapter 4. HASEX partially realizes some of the core features of the VA system, which provides (i) a resource management system with scalable renting capability over a selection of nodes from multiple resource providers and (ii) a cost-effective system that offers high resource availability, high utilization, performance, and scalability at lower cost, and the opportunity for incremental investment and immediate return. Through replicated experiments, we showed how the HASEX prototype supported by our VA framework performed comparatively better than the conventional gLite Grid middleware (Codispoti et al., 2010) in terms of delivering high QoS satisfaction and low resource cost.

8.1.2 Costing Framework

8.1.2.1 Cost-benefit-Analysis of Distributed Computing Infrastructures.

The Grid computing vision promises to provide global access to computing resources for demanding distributed and parallel high performance computing (HPC) applications. However, as we have discussed often in the thesis, the performance (i.e., the response times) achievable when using the Grid systems has not been very satisfactory. It may therefore be easier and cheaper for an individual user to build a private resource system consisting of smaller amount of nodes to serve their high performance computing needs. However, such a system may not be powerful enough to accommodate the high requirements of adaptive and interactive applications that have complex computation workloads with sudden surges in demand. Alternatively, a user may choose to purchase and maintain a large dedicated HPC system (e.g., up to 1000 nodes), in order to meet the demand for high performance computing and to have sufficient control over its use. However, this option could be far more expensive.

In Chapter 5, we have carried out a cost-benefit analysis of a small private resource computing facility consisting of 32 nodes and a large dedicated HPC computing facility with 1024 nodes. Using real cost data collected from both the LUNAR project and the LLNL computing facility, we carried out total cost of ownership (TCO) analysis of the two systems. First, we used the financial expense data obtained from other researchers and we detailed the specific costs of targeted distributed computing infrastructures: a private resource system for the LUNAR project (32 nodes, 128-processor system) and a dedicated HPC system for the Lawrence Livermore National Lab, USA (1024 nodes, 4096-processor system). Second, we used the financial data from the EGEE Grid project (a global Grid infrastructure with an approximate total of 100,945 processors) to determine the real cost of renting resource nodes from a large-scale Grid production system.

Based on our estimated calculated cost and using real workload traces collected from a LLNL HPC system, we explored the performance and cost effectiveness of a VA system that has the ability to rent processing power based on workload demand from existing Cluster and Grid systems. From our analysis, we demonstrated that there is a potential cost-benefit in adopting a small VA system compared to building and maintaining a small private system or a large dedicated HPC system. The VA approach can achieve higher QoS satisfaction compared to a small private resource system, while at the same it incurs significantly lower resource cost when compared to a large dedicated HPC system. This important finding has validated the feasibility of our VA approach to provide a new avenue for agility, service improvement and cost control in comparison to a conventional resource management approach where there is no rental mechanism.

8.1.2.2 Costing Model

A cost-effective framework implies that the resource management system must aim to improve the overall performance, scalability and cost manageability of the large-scale resource sharing infrastructure. Therefore, there is a strong need for the mechanism to satisfy applications' QoS requirements while lowering the cost associated with renting computing resources. Based on the performance model in Chapter 5, we further introduced a costing model in Chapter 6 that provides a more comprehensive model that the VA can use to quantitatively measure the effectiveness to deliver satisfaction of users' QoS requirements while keeping the rental associated cost to the minimum. The costing model is formulated based on an economic approach. It captures revenues generated from users and also expenses incurred from renting resources. The rental expenses derive from the need to deploy rented nodes from external providers, and the operational costs to maintain them. These include the administrative cost (i.e., deployment overhead and network cost) and other operational costs (i.e., electricity, personal staff, floor spaces etc). Applications express the monetary value of their jobs as the price they will pay to have them run, and the gap between this monetary value (revenue), the penalty for quality of service (i.e., deadline) violation and the resource expenses to run the job is simply the job's profit. The profit provides a useful single evaluation metric that captures the trade-off between earning monetary values, penalty for deadline violation, and paying for the resource cost.

8.1.3 Rental Policies

The provision of effective rental policies is essential for the economic viability of the VA system. In Chapter 6, we have proposed several aggressive and conservative policies that operate in a reactionary mode, where a VA initiates rental requests for additional nodes when there is a sudden increase in demand or when the nodes are running low. Taking into account the additional parameters of execution deadlines and virtual currency (i.e., monetary values), we further presented cost-aware rental policies that can significantly enhance the rental decision-making process. Based on real experimental workloads collected from the LLNL HPC system, we explored the impact of the cost-aware policies on various job deadlines, monetary values, system revenue and system profitability. We also examined how load, job mix, job values, job deadlines, node heterogeneity, rental duration, node lead time, job sizes, and rental price influence the system's profit. We further examined the impact of uncertainty of demand, uncertainty of resource availability and uncertainty of charges made by resource providers. Our work is the first step towards the evaluation of the impact of a rental-based system on various rental options and under various workload conditions.

8.1.4 Service Level Agreements

The lack of support functionality such as the lack of SLA management and the difficulty in providing QoS management in a distributed environment means that many of today's resource management systems operate on a best-effort basis. Best-effort systems are characterized by a lack of service level guarantees, and the quality of service they provide may vary over time due to workloads, resource availability, etc. In Chapter 7, we have proposed an SLA management (SLAM) framework which is built upon an extension of our VA framework to promote long-term rentals using service level agreements (SLAs), or long-term contracts. Such an extension promotes long-term capacity planning, and enables the VA to plan rental decisions in a proactive mode, rather than on a reactionary basis (i.e., unlike the conventional rental policies which we have proposed in chapter 6.

The extended SLA framework offers the facility for users to specify additional resource information such as the total load, the estimated average number of requested nodes, the estimated average job deadline, the estimated average penalty for non-compliance and the total monetary values (TMV) for the whole period of application execution. Hence, rental decisions can be effectively improved because long-term capacity planning can be anticipated in advance by using the additional information provided in the long-term SLA contract. Based on the SLAM framework, we presented SLA-aware policies that make use of both individual job information and the SLA contract information to perform more effective scheduling and rental decisions. The policies were extensively evaluated using real workload traces from the LLNL computing facility to demonstrate their applicability and effectiveness in improving the VA's ability to meet QOS requirements and lower resource cost (i.e., to improve overall profit).

8.2 On-going and Future Work

8.2.1 Scheduling and Rental Policies

In this thesis, we have proposed a number of complementary approaches, which we have brought together to form a cost-effective resource management framework for distributed computing. All of these are worthy of further exploration either independently or as an integrated set. In many ways, this thesis has only scratched the surface with regards to rental policies and SLA-aware mechanisms that can be exploited under a VA-like environment.

Beginning with the rental mechanism, we are exploring the means of refining the rental algorithms and policies to improve the overall profits. Although we have obtained extremely good results, we intend to explore the use of stochastic approaches to further improve the rental decisions under various workload conditions and rental options. The stochastic approaches will take into account random events such as realistic random demands, resource breakdowns, and resource uncertainty. It is envisaged that this will allow us to identify resource request patterns that repeat over time, which can be helpful to optimise rental decisions.

In addition, we intend to conduct an exploration of how advance reservation and backfilling techniques can be incorporated within the VA environment. So far, we have only investigated simple queuing policies such FCFS, SJF (Shortest Job First), EDF (Earliest Deadline First) etc., but it is also important to investigate the impact of more sophisticated scheduling algorithms such as backfilling and sophisticated advance reservation techniques on the VA. We also intend to examine the impact of other factors that we have not investigated in detail in this thesis. For example, the other factors may include storage and bandwidth availability and constraints, which must be considered when dealing with the transfer of large data files.

8.2.2 HASEX Resource Management

The design and the implementation of the HASEX proof-of-concept prototype only partially solves some the key features of a VA framework. There is much further development work that needs to be carried out. For example, the pricing service has not been fully implemented in the current HASEX prototype system.

In the context of HASEX, further development work is needed to implement standard interfaces and protocols that would enable the required interoperability between Grid and Cloud environments. HASEX will need to overcome the interoperability challenges often inherent in traditional Grids and Clouds by implementing a very strict separation of concern and minimalistic interfaces, and reducing cross-site concerns to a minimum. HASEX also needs to incorporate pricing schemes (Caracas and Altmann, 2007) and increase flexibility to support different billing schemes and accounting for services with indefinite lifetimes as opposed to finite jobs, with support to account for utilization metrics relevant for virtual machines. Moreover, HASEX needs to be compliant with the Web Services Resource Framework (WSRF) (Foster et al., 2005) specification, which is the specification language for Web Services that support rapid deployment of heterogeneous resources from both Grids and Clouds. A specified WSRF resource requirement should be converted by the HASEX into submission languages that are recognized by various Distributed Resource Managers (DRMs). Currently, the HASEX system only provides a simple requirement definition method based on XML for the HASEX Rental API.

There are many other areas of future work that are related to HASEX. For example, extensions to other Cloud systems such as Amazon EC2 (Inc, 2008), Eucalyptus (Nurmi et al., 2008) and Globus Nimbus (Martinaitis, Patten, and Wendelborn, 2009) are advocated and therefore standard APIs need to be implemented to abstract the interaction with Cloud systems. Furthermore, HASEX should be hypervisor neutral; the current implementation only supports Xen hypervisor, but it needs to support KVM (Kivity, 2007) and VMWare (Sugerman, Venkitachalam, and Lim, 2001) hypervisors as well. Moreover, HASEX will need to offer support for various distributed file systems in order to serve data-intensive applications. Currently, future work is planned to support both centralized (e.g., NFS (Zhao, Zhang, and Figueiredo, 2006)) and decentralized (e.g., PVFS (Yang et al., 2005) and (Bockelman, 2009)). With respect to security, we also aim to explore in future work how secure Virtual Private Network (VPN) (Wolinsky, Liu, and Figueiredo, 2009) gateways can be implemented on the HASEX system to allow the VA to communicate and negotiate confidentially with resource providers for SLA agreements over a public network via authenticated and trusted communication links.

8.2.3 Service Level Agreement

More research work on the design of a sophisticated admission control is also required. Currently, the thesis makes the assumption that the admission control always accepts a new SLA contract. However, a more sophisticated admission control could be developed so that it takes account of resource availability when determining whether or not to accept a new SLA contract. For the charging model, we have made a simple assumption that the monetary value is normally distributed within the ratio of the means for each parameter's high-value and low-value. The difference of the means between the low-and high-value jobs are stretched by a monetary value factor. However, this simple model of assigning monetary values may not be appropriate in some situations. For example, the monetary value to be assigned can vary depending on the VA operating condition. Furthermore, the charging function can be adaptive in accordance to the supply and demand of resources. For instance, if demand for a resource node is high, the job can be charged at a higher cost so as to prevent users from overloading the system at that particular time, and this would also maintain the equilibrium of supply and demand of resources. We intend to further explore the pricing issues to derive suitable algorithms for determining the optimal monetary value to assign for each job.

8.2.4 Market Dynamics of Supply and Demand

The costing model relies on the market dynamics of supply and demand (penalty cost vs. rental cost) to provide the optimal QoS for the price users are willing to pay in terms of monetary values. However, we have not explored how a market equilibrium may be achieved and how setting certain rental rules and policies would affect pricing. A simulation of such demands from a distributed system like EGEE could provide a better understanding of how these convoluted aspects interact with each other. We therefore aim to explore how resource price can reach equilibrium using market modelling techniques such as Game theory (Wei et al., 2009; Shang et al., 2010; Teng and Magoulès, 2010; Pal and Hui, 2011) for future work. Furthermore, it is useful to know the effects of monetary values on an application's ability to run jobs. For example, what stops two applications (app_a and app_b), which have the same amount of monetary values and the same urgency to use resource *R*, from outbidding each other until they use all their own credit and thus driving the price for Resource *R* artificially high for some time? Further investigations are required to investigate such behaviours.

8.2.5 Resource Discovery

Another key aspect of the VA is the ability to discover resource providers without relying on a centralized global meta-scheduler, and the ability to discover the most suitable resources at a reasonable cost in the shortest time. Resource discovery activities involve searching for the appropriate resource types that match the user's job requirements. To accomplish this goal, a resource discovery system that supports the desired look-up operation is mandatory. Various kinds of solutions to this problem have been suggested, including the centralized and hierarchical information server approach (Czajkowski et al., 2001; Zhang, Freschl, and Schopf, 2003; Podhorszki and Kacsuk, 2003; Pan et al., 2007).

Traditionally, resource management systems such as gLite (Codispoti et al., 2010), Condor-G (Frey et al., 2002), and Nimrod-G (Buyya, Abramson, and Giddy, 2000) use the services of centralized information services such as (Podhorszki and Kacsuk, 2003), Hawkeye (Pan et al., 2007), MDS-1 (Czajkowski et al., 2001) to index resource information. Under centralized organization, the meta-scheduler sends resource queries to a centralized resource indexing service. Similarly, the resource providers update the resource status at periodic intervals using resource update messages. This approach has several design issues: (i) it is highly prone to a single point of failure; (ii) it lacks scalability; (iii) it has high network communication cost at links leading to the information server (i.e., network bottleneck, congestion); and (iv) the machine running the information services might lack the required computational power required to serve a large number of resource queries and updates (Ranjan et al., 2007). A hierarchical organization of information services has also been proposed in systems such as MDS-3 (Zhang and Schopf, 2004) and Ganglia (Massie, Chun, and Culler, 2004). MDS-3 organizes Virtual Organization (VO) (Alfieri et al., 2005) specific information directories in a hierarchy. A VO includes a set of organizations that agree on common resource sharing policies. Every VO in the Grid designates a machine that hosts the information services. However, this approach also has similar problems to those of the centralized approach such as one-point of failure, and it does not scale well for a large number of users/providers.

To eliminate potential performance and reliability bottlenecks, there is a strong need for decentralized resource discovery without the overhead of a global knowledge about the Grid condition; decentralized resource discovery would ensure fast responses to resource renting requests. Some research efforts have been made that recognize the need for decentralized resource discovery solutions. For instance, Das and Grosu (2005b) explore a decentralized combinatorial auction that can assist the service provider to discover resources without relying on a global broker. Such an auction system performs resource discovery by choosing the bidder with the lowest cost criterion. More recently, Iordache et al. (2007a); Fiscato, Costa, and Pierre (2008); Costa et al. (2009); Ranjan and Buyya (2010) have proposed peer-to-peer solutions that resolve the issues associated with discovery of available resources. Such solutions could potentially allow resource discovery to be carried out with only very partial knowledge about the Grid as a whole. Their works are complementary to our work and they are worthy of further exploration in future work.

8.2.6 Virtualisation and Cloud Computing

As a final note, it is important to mention the emergence of the Cloud Computing paradigm (Sullivan, 2009). The concept borrows many principles and concepts established by the Grid community to provide a generic framework for the dynamic provision of services over the Internet. With Cloud computing, organizations can lease infrastructure resources in the form of virtual machines, removing away concerns regarding the technology infrastructure that provides them, and allocating and de-allocating physical resources at will to meet the overall demand. For this to be possible there are numerous challenges that must be met in the domain of resource management, accounting, security, etc., many of which are being actively explored by the research community.

An important concept in Cloud computing is that of elasticity: the dynamic scaling of allocated capacity based on application state. An example of this is the ability to allocate additional resources as the

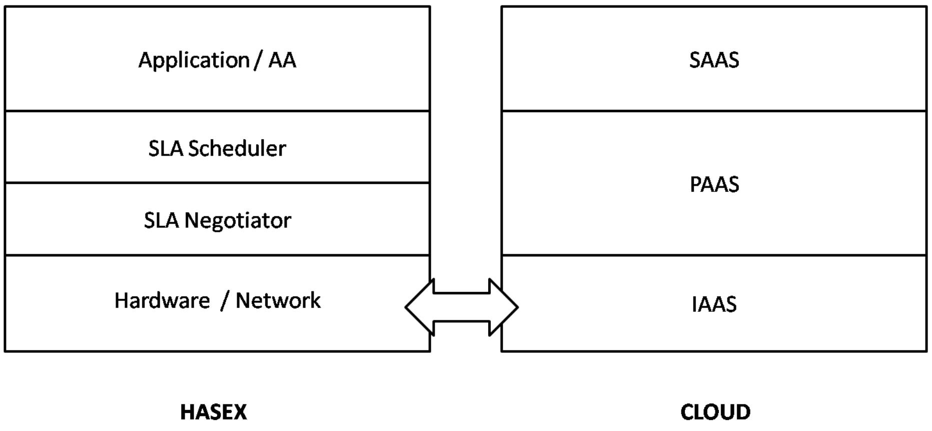


Figure 8.1: A conceptual diagram on the incorporation of HASEX with Cloud computing.

application workload increases (Napper and Bientinesi, 2009). The use of rental and SLA mechanisms as explored in this thesis can complement the monitoring of application-level workloads in order to anticipate future resource requirements and minimize over-provisioning. Cloud computing can promote the use of open standards and platforms with the ultimate goal of enabling a federation of Clouds across multiple providers. The notion of Cloud computing has numerous interesting similarities and properties in common with our VA framework such as the ability to rent resources from resource providers and the ability to adapt to resource infrastructure dynamically.

Figure 8.1 illustrates the hierarchical layers of Cloud Computing. The lower level of Cloud computing is Infrastructure-as-a-Service (IaaS), which describes platforms that offer computing infrastructure typically provided as a virtualization environment. The IaaS platform would provide the ability for consumers to scale their infrastructure up or down according to demand and pay for the resources consumed. This is where pre-configured hardware is provided via a virtualized interface or hypervisor. At this layer, there is no high-level infrastructure software provided such as an operating system, and users are required to embed their own virtual applications. Above the IaaS layer, the Platform-as-a-Service (PaaS) provides the ability for building and deploying custom applications on Cloud resources. Furthermore, the Software-as-a-Service (SaaS) layer resides on top of the PaaS layer and offers fully functional applications on demand to provide specific services such as email management, web conferencing and an increasingly extensive range of other commercial applications.

In future work, we aim to build a communication bridge between the hardware/network layer in HASEX and the IaaS layer in Cloud (as shown in Figure 8.1). The idea is to integrate HASEX with the IaaS Cloud infrastructure so that a trust relationship can be established via SLA agreements between both parties. The SLA agreements would enable HASEX to rent resources from Cloud providers in times of high demand. Ultimately, this provides the ability for HASEX to offer computational or processing capacity to users using Cloud resources and HASEX could become a uniform interface to all computational resources.

8.2.7 Chapter Summary

The scale and operation of the resource infrastructure across organizational boundaries requires novel solutions to effectively exploit collections of independent computational resources because current resource management infrastructures suffer from a wide variety of problems, which include scalability bottleneck, resource allocation delay, limited quality-of-service (QoS) support, and lack of cost-aware and service level agreement (SLA) mechanisms. What we have achieved in the context of this thesis is the design and implementation of a cost-effective framework which enables organizations to construct their own VA systems which are independent from competitors. Resources are rented based on the workload demand and they are negotiated independently from multiple resource providers. As far as scheduling is concerned, the VA is independent from competitors and can instead concentrate on managing its own resources. As a result, the VA offers a scalable resource management solution with minimal infrastructure and operating costs.

In an environment where resource providers may have very different interests and objectives for contributing resources, the cost-effective framework ensures that resource autonomy is preserved through fully decentralized operation and is adaptive in nature, relying on rental and SLA mechanisms to maximize the quality of service for the end users and also to protect resource provider interests. Computational resources can now be aggregated in a regular and ad-hoc manner to form federated resource infrastructures that reflect the needs of both providers and users. The VA framework works alongside current distributed systems (such as Cluster and Grid systems) to ensure a seamless integration. The end-result proves to be both cost-effective and scalable and the approaches have been evaluated through a proof-of-concept implementation and through extensive simulations.

Appendix A

Simulation Framework

Evaluating the efficiency of rental mechanisms and resource allocation strategies is not always possible in a production-level resource management environment: developers may lack administrative control over the systems in use or there may be policies that cannot be modified without affecting existing users. Setting up an independent test-bed can be costly and, in addition, the effects of policy changes are often only observable over long periods of time. It is for these reasons that relying on simulation frameworks is often preferable or even a necessity. The evaluation of rental mechanisms requires the ability to perform controllable and repeatable experiments with tight control over workloads, which is difficult to achieve in highly dynamic environments. The need to replicate experiments in a systematic manner also means that we must be able to reproduce these workloads on demand. It is not always the case that we have access to the applications and data on which users may rely.

Simulation tools offer a flexible and scalable approach to create reasonably detailed resource management models, and to repeat and control target experiment conditions. They also provide an easy way to test and evaluate a variety of different rental- and SLA-based policies and algorithms, and offer a convenient approach for varying system and workload parameters to better understand performance under a wider variety of conditions than may be possible in real systems with real implementations. Therefore, in the context of the work described in Chapter 5, Chapter 6 and Chapter 7, we have used and extended the Concatenated LOcal and Wide Area Network (CLOWN) (Sorensen and Jones, 1992) discrete-event simulator for experimental purposes. In this appendix, we describe the CLOWN system architecture and the process by which our Virtual Authority (VA) system is designed and extended in CLOWN for experimental purposes.

A.1 CLOWN Simulator

The Concatenated LOcal and Wide Area Network (CLOWN) (Sorensen and Jones, 1992) is an object-oriented simulation environment written in C and C++. This section discusses the system architecture of CLOWN (shown in Figure A.1) in relation to the process of developing a resource management model. The CLOWN simulation model creation has two parts. The first involves the creation of user-defined modules or modifying built-in network object libraries.

A CLOWN module defines how a module executes events through function calls. It would follow

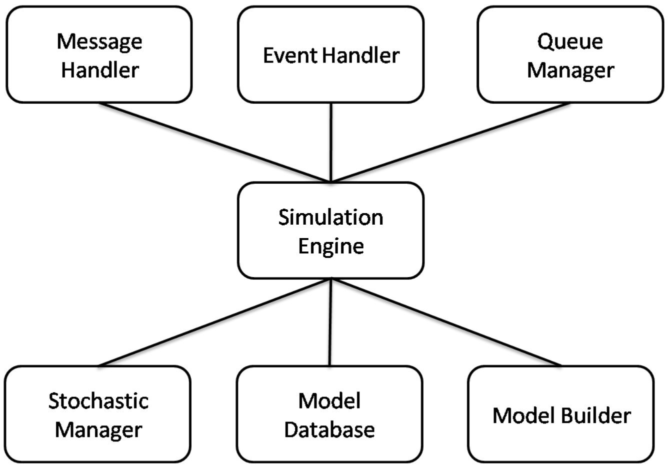


Figure A.1: CLOWN system architecture.

that each simulation event type has a corresponding handler function in the module. Once defined, all these modules are then loaded into the Model Database. Next, CLOWN accepts an input file describing how the network model is built up from the modules in the Model Database. CLOWN parses the file and creates the model in memory through the Model Builder. The Simulation Engine then checks the file for validity using the entries in the Model Database. Once this has been verified, simulation can begin. Every simulation model is a specification of a physical system in terms of a set of states and events. CLOWN employs an Event List which chronologically stores states. Simulation progresses when the element at the front of the list is triggered, causing states to change and/or add new events to the event list. These are handled by the Event Handler subsystem.

The subsystems of CLOWN are as follows:

- Message Handler – facilitates information exchange between network modules;
- Queue Manager – a library of different queue types (FIFO, Shortest Job First, Earliest Deadline First etc.) for use by network modules;
- Stochastic Manager – a library of different random number generators that return numbers according to a particular distribution. Clown provides a library for exponential, Weibull, uniform, and normal distribution.

A.2 Simulation Engine

As noted earlier, CLOWN modules and events are controlled by the simulation engine. The simulation engine controls simulations by continuously processing events in virtual time order from a collection of inter-connected modules. Figure A.3 shows how the simulation engine drives simulations by generating new events and providing module-specific event handling code. They use encapsulation to hide details of their implementations. State information is exposed solely through the posting of relevant state information on the associated event table. External access to module functionality is similarly limited.

We have extended CLOWN with user-defined modules for our resource management system: Workload Generator, VA Scheduler, VA Negotiator and Resource Providers. From the point of view of the

```
void CLOWN::ExecuteEvent( CLevent* ev ) {
switch( GetCodeFromDescription( ev->GetDescription() ) ) {

        case GENERATEWORKLOAD:          // Call the functions dealing with this event.          ↩
             GenerateWorkload( ev );             //simply place some instructions here.
                   break;
        case GENERATECONTRACT:          // Call the functions dealing with this event.          ↩
             GenerateContract( ev );
                   break;
        case TASKCOMPLETION:            // Call the functions dealing with this event.          ↩
             TaskCompletion( ev );
                   break;
        case JOBCOMPLETION:             // Call the functions dealing with this event.          ↩
             JobCompletion( ev );
                   break;
        case SCHEDULERPERIOD:           // Call the functions dealing with this event.          ↩
             scheduler_period( ev );
                   break;
        case LEASEEXPIRED:              // Call the functions dealing with this event.          ↩
             LeaseExpired( ev );
                   break;
        case LOCALSCHEDULER:            // Call the functions dealing with this event.          ↩
             scheduler( ev );
                   break;
        case SERVE:                     // Call the functions dealing with this event.          ↩
             serve( ev );
                   break;
        case ALLOCATE:                  // Call the functions dealing with this event.          ↩
             allocate( ev );
                   break;
        case RENTAL:                    // Call the functions dealing with this event.          ↩
             rental( ev );
                   break;
        case RENTARRIVAL:               // Call the functions dealing with this event.          ↩
             update_inventory( ev );
                   break;
        case STATISTIC:                 // Call the functions dealing with this event.          ↩
             report_statistic( ev );
                   break;

        default: fprintf( stdout, "CLOWN: Unknown event: %s\n", GetCodeFromDescription( ev->↩
             GetDescription() ) );          exit( -1 );
}
} // END void MasterObject::ExecuteEvent( CLevent* )
```

Figure A.2: Sample listing of main events used in the simulator.

engine, modules (Workload Generator, VA Scheduler, VA Negotiator, Resource Providers) are simply black boxes that contain event queues and have associated event handling code. The details of how they handle those events is entirely dependent upon the specifics of those particular instances. Similarly, details on when and which events to generate are also dependent upon the specifics of module implementations. Figure A.2 sshows a sample listing of events used in the simulator. For each event listed (e.g., `GENERATEWORKLOAD`, `GENERATECONTRACT` etc.), its corresponding function will be invoked when the simulation engine triggers it. The descriptions for these events are also listed in Table A.1.

At each step of the simulation, the task of the simulation engine is to dequeue the event with the lowest virtual time from some module's event queue, invoke module-specific event handler code for that event (which may create and enqueue new events on other module event queues), and optionally route that event to a destination module and enqueue it on its event queue (e.g., routing a `GENERATEWORKLOAD` event from the Workload Generator module to the VA Scheduler module). The simulation ends when a stopping condition is reached. This could occur when there are no events left or when the simulation has reached its defined end time, or when all jobs have executed to completion.

Events	**Descriptions**
`GENERATEWORKLOAD`	Generates and submit new Jobs to the VA Scheduler.
`GENERATECONTRACT`	Generates and submit new SLA contracts to the VA Scheduler.
`TASKCOMPLETION`	Removes task from its resource node and updates the resource availability information.
`JOBCOMPLETION`	Removes job from the Queing system and update statistic.
`SCHEDULERPERIOD`	Triggers the system to traverse all queued jobs from the Queing system.
`LEASEXPIRED`	Removes rented node from the system and computes the total rental cost.
`LOCALSCHEDULER`	Inserts new job arrival into the Queing system.
`SERVE`	Serve and removes the job from the Queuing system if there are sufficient nodes.
`ALLOCATE`	Allocate scheduled job to resource nodes.
`RENTAL`	Rents additional nodes upon new arrival job or when a rental condition event has been triggerred.
`RENTARRIVAL`	Updates resource availability with newly rented nodes information.
`STATISTIC`	Updates performance metrics when a job has completed its execution or when a rental lease has expired.

Table A.1: Descriptions of the main events used in the simulator.

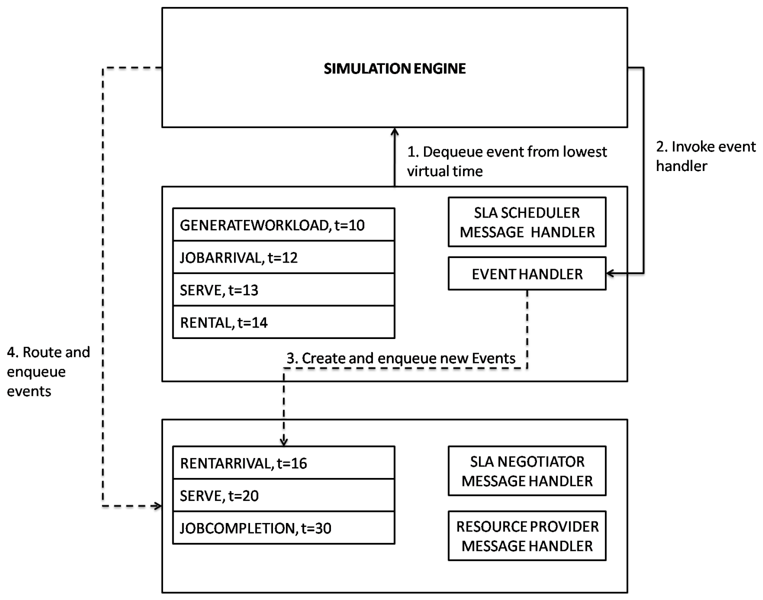

SIMULATION ENGINE
1. Dequeue event from lowest virtual time
2. Invoke event handler
GENERATEWORKLOAD, t=10
JOBARRIVAL, t=12
SERVE, t=13
RENTAL, t=14
SLA SCHEDULER MESSAGE HANDLER
EVENT HANDLER
3. Create and enqueue new Events
4. Route and enqueue events
RENTARRIVAL, t=16
SERVE, t=20
JOBCOMPLETION, t=30
SLA NEGOTIATOR MESSAGE HANDLER
RESOURCE PROVIDER MESSAGE HANDLER

Figure A.3: Event-driven simulation in CLOWN.

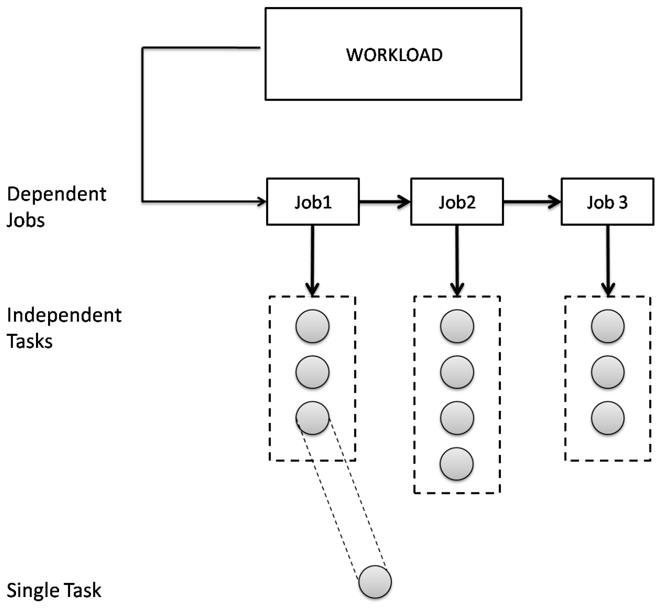


Figure A.4: Levels of relationship of jobs.

A.3 Workload Generator

The Workload Generator generates the jobs and the resource requirements to be submitted to the VA Scheduler. It generates information about jobs, their structure, resource requirements, relationships between modules, time intervals etc. The simulator is a model in which each job consists of one or more tasks. A job may also contain preceding constraints between tasks. The relationship between a set of tasks is described through a Directed Acyclic Graph (DAG) or a workflow using the task graph builder. The workload consists of job parameters such as job ID, number of tasks (job size), processing unit of each task, and tasks value. In some cases (for advance reservation scheduling), task estimated time is specified as an additional parameter. These parameters are used by the scheduling policies when performing scheduling. Figure A.4 illustrates the level of relationship of jobs.

To summarize, for each job, the following parameters are specified:

1. Job ID;
2. Number of nodes requested per job (job size);
3. Processing unit for each task within a job;
4. Total job runtime (the job runtime is calculated based on the load and the processing node);
5. Completion deadline;
6. Job value.

For instance, suppose a distributed application that follows the process task farming model (all independent tasks). This is modelled with the following parameters:

1. number of jobs ≥ 1

2. $n = 1$ task per job or job size$= 1$.

If a distributed application consists of parallel jobs and multiple dependent tasks, the parameters are defined as follows:

1. number of jobs ≥ 1

2. $n \geq 1$ tasks per job or job size≥ 1,

Figure A.4 shows the level of jobs and tasks in more detail. Job 1 consists of three tasks, Job 2 consists of four tasks, and Job 3 has three tasks. However, notice that Job 2 cannot execute until Job 1 has finished its execution (i.e., all tasks for Job 1 must have finished execution before Job 2 can start). Similarly, Job 3 cannot execute until all tasks of Job 2 have finished their execution.

There are primarily two different types of applications of interest:

1. Best Effort (non-urgent) and

2. Interactive (urgent).

A Best Effort (BE) application does not need immediate access to resources because the time it takes to complete the job is not crucial. This type of application is not interactive because users do not expect instantaneous feedback of the execution results. Therefore, they normally have flexible deadlines as part of their job specifications. The typical information that is made available to the scheduler for BE-type applications is the number of node(s) required and the user's estimated completion time.

Conversely, interactive applications are more demanding because they need instantaneous feedback of the job execution results because users expect to view them on an interactive basis. The requirement for this type of application is to be able to serve a job within certain time units (termed the deadline). The job execution result can be useless if the job is not completed within its deadline. However, some applications of this type are not particularly strict in this regard. The deadline can either be hard or soft; a hard deadline needs to be accommodated within the time deadline given because if it is not met, the job execution result becomes useless. Alternatively, a soft deadline provides an indicator to the scheduler about when the job is expected to be completed. Failing to meet this deadline does not mean that the job execution results become useless, but the failure to meet such deadline diminishes user satisfaction.

The simulator consists of different types of information parameters: workload parameter, scheduler parameter and system parameters. The workload characteristics include arrival process, size distribution, and runtime distribution which are generated randomly. These workload characteristics are normally obtained from workload traces and models which are based on workload logs collected from large-scale parallel systems in production use (e.g., the Lawrence Livermore National Lab log).

A set of parameters is also used by the simulator to calculate statistics. These parameters include job count, arrival rate, load factor, task count, task service time (runtime), and simulation time etc. The complexity and accuracy of these distributions range from simple Poisson processes, uniform, exponential, and gamma distributions to more complex distributions based on actual workload trace data. For each generated parameter, the following probabilistic attributes and constraints can be specified:

- Mean value
- Standard deviation (stdev)
- Minimal and Maximal Value (min, max)
- Seed for random process (seed)
- Distribution e.g. uniform, normal, poisson, exponential, gamma (probabilistic distribution)
- Time period (current time, start time and end time)

Experimental performance evaluation studies require representative workloads to produce dependable results. The most important aspect of our study is to examine how the deadline parameters could influence the system performance. Unfortunately, previous studies give no guidance on how applications or users prioritize their tasks, since no traces from deployed user-centric parallel scheduling systems are available. This is unfortunate because the distribution of the maximum task values and penalties has a significant impact on the results. Therefore, a synthetic workload is used to create a set of combinations for the deadline parameters to cover several possibilities. We adopt a similar methodology in Irwin, Grit, and Chase (2004) to model the urgency and completion deadlines parameters through two classes: (1) high urgency and (2) low urgency (Yeo and Buyya, 2007).

Both the urgency and completion deadlines are normally distributed within high and low classes: unless otherwise specified. For example, to represent a higher fraction of urgent requests, 20% of requests can be assigned with a low deadline *deadlinefactor*runtime* and 80% of the requests can be assigned with a high deadline *deadlinefactor*runtime*. This model is realistic since the AA that submits a more urgent task expects to finish earlier compared to a non-urgent task. Similarly, the same also applies to the urgency deadline as a more urgent request will expect to receive nodes earlier compared to a non-urgent request (which could tolerate longer delays).

The level of urgency between jobs can also be varied using the urgency factor to differentiate between urgent and non-urgent jobs. For example, with a deadline distribution factor 10, urgent jobs are defined as those with deadlines of average 10T (time units), while non-urgent jobs have deadlines of average 100T.

The same also applies for job execution time and job size parameters. For instance, the average running time of long and short jobs can also be varied by a distribution factor. For example, with a distribution factor of 10, long running jobs are defined as those with average running time of 100 time units, while short running jobs have average running time of 10 time units.

The job size can also be differentiated. For example, a large job size can be defined as a job with greater than 16 nodes, while a short job size can be defined as a job with less than 16. Similarly, the job computation complexity of short and long execution can also be described using a ratio based on their task lengths (task processing units).

Using a variation of the parameters above, we can create a wide range of parameters for our synthetic workloads under different simulated environments. A set of sample workloads is given below.

Workload	Urgency Level (%) Urgent : Non Urgent	Job Size (%) Small : Large	Computation (%) Long : Short
W1	0 : 100	80 : 20	80 : 20
W2	20 : 80	20 : 80	80 : 20
W3	80 : 20	20 : 80	20 : 80

Table A.2: Sample simulator parameter-settings.

Workload 1 (W1): Environment where the majority of jobs are smaller by job size, but where jobs with large job size constitute most of the computation time in the system.

Workload 2 (W2): Environment where the majority of jobs are long-running independent jobs and there are fewer short-running interactive jobs.

Workload 3 (W3): Environment where the majority of jobs are interactive and there are a few best effort jobs with a large job size.

A.4 Application Agent

The Application Agent (AA) module can replace the Workload Generator if developers intend to simulate models of the internal structure of a parallel application, in order to be able to investigate the connections between application behaviour and scheduling. In all our experiments, all jobs are based on the workload traces we have chosen. However, the simulator also supports the application agent module if simulating application execution characteristics is necessary.

The application agent (AA) module comprises a number of sub-modules such as the task preparer, task distributor, task controller, and task manager. A job is submitted by the application agent with a set of parameters that describes the job specification/requirement. The AA translates the job request into a resource request and establishes communication with the scheduler. The scheduler receives the request, places it in the queue and the scheduler determines either to serve the current request or any other request currently held in the queue. This decision will be based on the scheduling scheme employed. If one of the request is served from the queue, the scheduler determines which node the request is to be allocated to. The request is scheduled immediately if sufficient node(s) are available. Otherwise, the task controller requests additional nodes from the Resource Providers via the VA Negotiator.

If additional nodes are requested, a non-blocking (i.e., asynchronous) event is triggered to detect the arrival of new nodes. Subsequently, the execution node is stored in the cluster node database upon node arrival. Upon allocation, a control process called the task controller is spawned to monitor the individual task(s) of a job independently. At the allocated node, the global session is established between the task controller and the node through the task manager. The task manager is spawned on the node to monitor task execution. The job executes and, upon its completion, its execution result is returned to the task controller. The task controller of the AA triggers a notification event to the application to notify that the execution result is ready to be accessed. Finally, the task controller triggers an event to indicate that the results are ready to be retrieved by the user. Figure A.5 illustrates the flow of information between the AA modules and the rest of the system modules. A more detailed description of the AA modules can be found in Liu, Nazir, and Sørensen (2009).

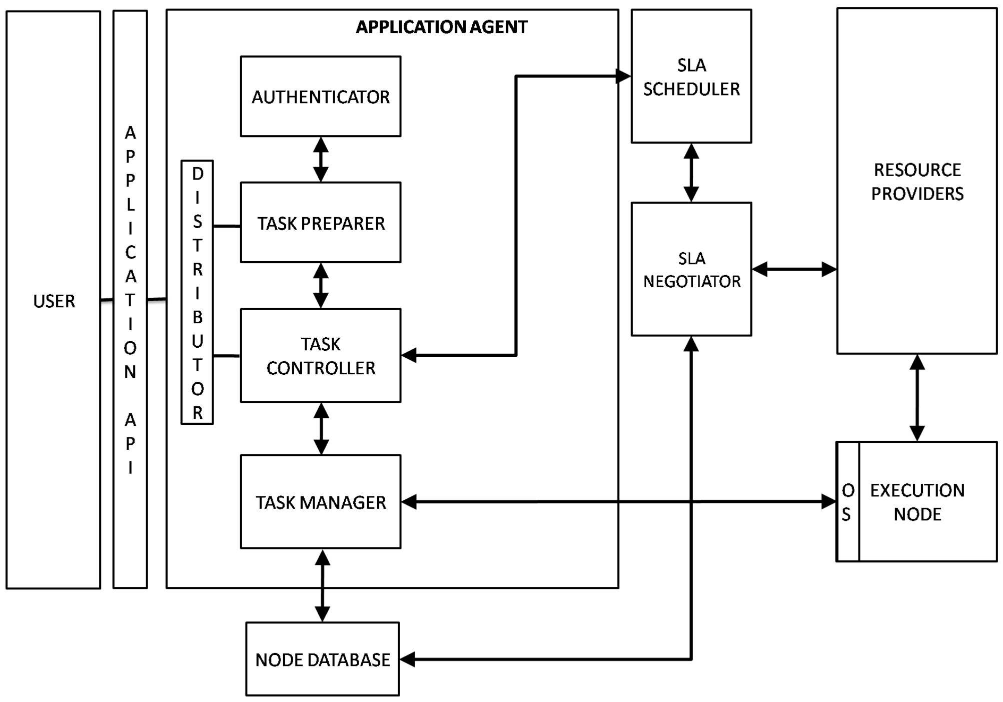


Figure A.5: The flow of information between the AA modules and the other modules.

A.5 VA Scheduler

The scheduling policies are implemented by the scheduler module. The core functionality is to schedule and assign arriving requests to computational nodes in the system. The aim of the scheduling policies is to determine the most efficient way to schedule jobs so that job waiting time and missed deadlines can be minimized. The scheduler makes use of the scheduling parameters such as deadline, job processing length, job estimated time, and job communication characteristics to make an efficient schedule. All incoming resource requests submitted by the AA will be sent to the scheduler module before they reach the other modules.

A queuing system is adopted for new arriving jobs. Jobs may have to wait before they can be scheduled, especially under a high load. Upon arrival of a job, the scheduler module first inserts the job into the queue. Different queue service disciplines are provided which include FCFS, shortest job size, shortest job first, and earliest deadline first.

Jobs may also have various levels of complexity depending on the application types. This could range from workflows, through large-scale parallel applications, to single tasks that require single resources. Depending on the types of jobs, scheduling strategies may require different parameters. For example, an advance reservation scheduling scheme would require an estimation of job execution time in order to perform reservation.

A scheduler is exposed to some information about incoming jobs. The information may include job requirements such as job size and job task length. However, the simulator can restrict the scheduler from having access to particular parameters, e.g., user runtime estimates, number of tasks etc. to suit specific scheduling strategies. How the scheduler makes use of the information would depend on the scheduling

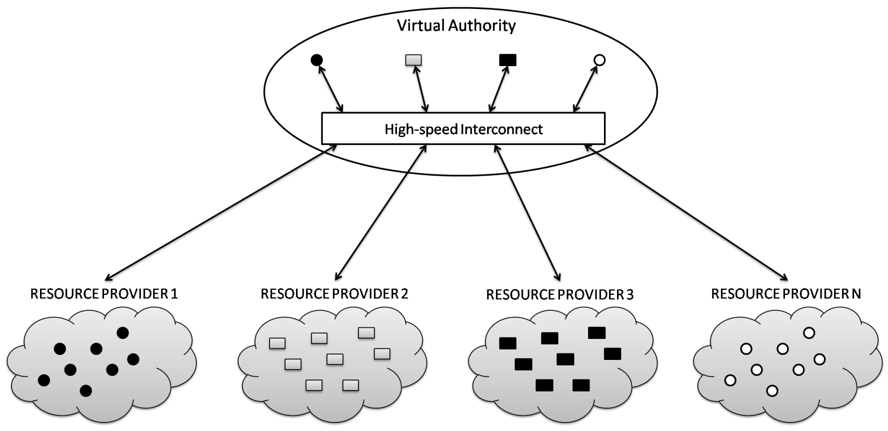


Figure A.6: A Virtual Authority consists of heterogeneous nodes with high-speed network interconnects which are rented from a group of Resource Providers.

strategy chosen.

A.6 VA Negotiator

The VA Negotiator implements the rental policies. Its role is to rent additional nodes if there is an insufficient amount of nodes to satisfy the current demand of arriving jobs. The rental policies can be categorized into two primary modes:

1. immediate mode and
2. periodic mode.

The immediate rental policy reacts whenever a job arrives, a job completes, or the number of available resources changes; it may chose to rent additional node(s), or it may decide not to rent. It may also choose to react when certain thresholds of specific-defined parameters have been reached, e.g., load, profits etc. Alternatively, the periodic rental policy reviews the resource level in the system on a periodic interval and determines whether to rent/release resources or do nothing. The main objective of both policies is to ensure that the system is kept to a suitable amount of nodes at the minimum cost to satisfy current and future demands.

A.7 Resource Providers

Resource Providers provide a group of homogeneous or heterogeneous resource nodes that are connected by high-speed interconnect. Figure A.6 shows how each resource node is represented by only one processor. A standard node is defined as the time it takes to process a standard task. On each node, the space-sharing (Feitelson and Rudolph, 1995) scheduling scheme is assumed whereby only one job/task is allowed to execute and run at any time. Any other queued jobs/tasks must wait until the currently scheduled job/task has finished execution.

Resource Nodes are negotiated and rented by the VA Scheduler. Once rented, they become part of the Virtual Authority (VA) (Figure A.6). All nodes that become part of the VA are available for job scheduling by the VA Scheduler.

A resource node consists of two parameters: processing speed and speed factor. A node can only be accessed by one task at a time. The completion time of a running task would depend on the node's processing speed. Therefore, we use the speed factor β to assess the capacity of all nodes against a standard node by defining a standard node and a standard task. Let t be the time it takes the standard node to process the standard task, e.g., 10T (time units). If speed factor β is set to one, the time it takes a computing node of type N to process a standard task is: $\tau = t/\beta_N$ e.g., 10=10/1.

A.8 Summary

The overview of the simulation framework in this appendix serves primarily to demonstrate that a suitable environment was created for the evaluation of rental- and SLA-based policies. We have extended the CLOWN simulator framework to perform the experiments in Chapter 5, Chapter 6 and Chapter 7 when it was not feasible to carry out the experiments on test-beds of real systems.

In order to obtain realistic simulation results with confidence we need to be able to do so in the presence of workloads that represent actual use. The need to replicate experiments in a systematic manner also means that we must be able to reproduce these workloads on demand. It is not always the case that we have access to the applications and data on which users may rely. Furthermore the scientific processes required by users may involve a number of different applications, the execution of which may have been automated by various workflow tools. Much of the information required to reproduce the loads imposed by these processes for the purpose of evaluating scheduling mechanisms can be extrapolated from the logs of real parallel workloads from production systems. Information about job submission times, running time, and job sizes will have been logged and made available to researchers interested in the evaluation of parallel systems, and specifically schedulers for such systems. By using the logs, jobs may be recreated with similar characteristics.

In this thesis, the Lawrence Livermore National Lab (LLNL) workload logs from the Workload Parallel Archive have been relied upon for the experiments presented in Chapter 5, 6 and 7.

Appendix B

The LLNL Workload Logs

The LLNL workload logs contain several months' worth of accounting records from a large Linux cluster called Thunder installed at the Lawrence Livermore National Lab. This specific cluster has 1,024 nodes, each node with four processors, which makes up a total of 4,096 processors. Each node boasts four Intel IA-64 Itanium processors clocked at 1.4 GHz and 8 GB of memory. The nodes are connected by a Quadrics (Petrini et al., 2002) network. When it was installed in 2004, LLNL was the second high-capacity supercomputing centre on the Top 500 list.

There are two versions of the LLNL workload logs: LLNL Atlas and LLNL Thunder. The LLNL Thunder log is considered a '"capacity'" computing resource, meaning that it was intended for running large numbers of smaller to medium jobs. In contrast, the LLNL Atlas log is considered a '"capability'" computing resource, meaning that it is intended for running large parallel jobs that cannot execute on lesser machines. The logs were graciously provided by Dr. Moe Jette from the Lawrence Livermore National Lab, who also helped with background information and interpretation.

B.0.1 Log Format

The original log is available as LLNL-Thunder-2007. However, the original log contains several flurries of very high activity by individual users, which may not be representative of normal usage. These were removed in the cleaned version. We therefore used the cleaned log for our experimental purposes and this log is available as a LLNL-Thunder-2007-1.1-cln standard workload format (swf) log from the Workload Parallel Archive.

This swf log file contains one line per completed job in the following format:

1. Job Number – a counter field, starting from 1.

2. Submit Time – in seconds. The earliest time the log refers to is zero, and this is the submittal time of the first job. The lines in the log are sorted by ascending submittal times. It makes sense for jobs to also be numbered in this order.

3. Wait Time – in seconds. The difference between the job's submit time and the time at which it actually began to run. Of course, this is only relevant to real logs, not to models.

4. Runtime – in seconds. The wall clock time the job was running (end time minus start time).

5. Number of Allocated Processors – an integer. In most cases this is also the number of processors the job uses; if the job does not use all of them, no information is provided.

6. Average CPU Time Used – both user and system, in seconds. This is the average over all processors of the CPU time used, and may therefore be smaller than the runtime. If a log contains the total CPU time used by all the processors, it is divided by the number of allocated processors to derive the average.

7. Job Size – an integer. The number of processors requested by the user to run the job.

Additional information such as User ID, Queue number, Used memory etc. are also available from the log, but we did not use them because they are irrelevant for our experimental purposes.

B.1 Data Analysis

After obtaining the simulation results, we carry out further statistical analysis on performance metrics (such as QoS satisfaction and resource cost). Statistical results based on multiple simulation runs improve the accuracy of the results. Each simulation experiment comprises a number of independent trials.

In a statistical data analysis, it is essential to determine the number of required independent runs. The Central Limit Theorem can determine the number of simulation runs required. Each simulation of this study is started by performing a small number of independent runs. Then we calculate the mean and standard deviation of the results that are used to determine the number of simulation runs necessary. An initial study found that a sample size of 10 should be sufficient. However, in order to provide better estimation, the number of simulation runs was set to be 20.

B.2 Summary

A realistic workload model is necessary for a realistic evaluation of our rental and SLA-aware mechanisms. We achieve this by using the LLNL workload logs which present full and realistic workload traces of distributed and parallel systems that capture many important characteristics of real workloads including job arrivals, job sizes, job runtime, and correlation between runtime and parallelism. These are all essential factors to consider in the studies of the performance of the rental and SLA-aware policies in this thesis.

Appendix C

IntelligentGrid Prototype

In Chapter 4, we presented the design, implementation, and performance of HASEX as an infrastructure for implementing Virtual Authority (VA) system. Using HASEX as the base infrastructure, this appendix presents IntelligentGrid, a prototype Grid system that enables a rapid provisioning of computational resources for immediate execution of high priority jobs in production Grid environments. Similar to HASEX, the basic idea of IntelligentGrid is to leverage the virtual machine (VM) technology to deploy additional Grid worker nodes (Codispoti et al., 2010) for the execution of serial and parallel jobs. The IntelligentGrid is based on the gLite Grid (Andreetto et al., 2008) middleware and the Xen (Barham et al., 2003b) hypervisor. Our preliminary experiments confirm the feasibility of the approach.

C.1 Introduction

A computational Grid enables the aggregation of multiple clusters for dynamic sharing, selection, and management of resources. Examples of Grid mid-dleware such as gLite (EGEE Project, 2007), Globus (Foster and Kesselman, 1997) and Unicore (Erwin, 2001) provide protocols and functionalities to enable these dynamic of sharing resources. Generally, they combine existing cluster management systems such as Condor (Thain, Tannenbaum, and Livny, 2005), SGE (Gentzsch, 2001a), LSF (Xu, 2001), and PBS (Henderson, 1995) job management systems to manage the computational needs of sequential and parallel applications.

Although much work has been done to improve the various aspects of Grid software, the current Grid and resource management systems have several shortcomings. First, it is often difficult to obtain as many resources as a job may need. In such situation, the user job is forced to wait in the queue until suitable resources are released by other running jobs that have completed their execution. In the worst case, the user job may not even run at all if the job requirements cannot be met by all accessible physical servers. Second, it is difficult to obtain the resources when one needs them at sudden spikes in demand. In particular, at high load, it is often difficult to find physical resources that exactly match the number of processors as well as software specification requested by the user. As a result, the job cannot be executed. Third, the number of resources that can be deployed at any given time is restricted to the number of physical machines; only one job can be executed on a single physical server node at any one time regardless of the number of processors available. Therefore, given the above limitations, there is a need to design a mechanism for on-demand deployment of resources when they are needed most; specifically, this implies that resources have to be made available immediately with very little advance notice. This requirement is especially crucial for high priority jobs.

We describe the IntelligentGrid, a system for rapid privisioning of Grid worker nodes for pending, urgent jobs. IntelligentGrid uses virtual machine (VM) technology to dynamically provision Grid worker nodes. The virtual machine technology is based on the Xen hypervisor (Barham et al., 2003b) and the Grid infrastructure is based on the well-known gLite middleware (EGEE Project, 2007). The gLite middleware is used by the Enabling Grids for the E-Science (EGEE) Grid (Berlich et al., 2006), which is currently the world's largest production Grid with collaborative efforts across 319 sites with a total number of 100,945 CPUs. Similarly, Xen (Barham et al., 2003b) is a popular open-source high performance virtualisation technology originally developed at the University of Cambridge. With the combination of both technologies and our proposed resource management, IntelligentGrid was developed to realize the need for the creation of customized Grid environment for the efficient management of users' tasks.

C.2 Related Work

Several efforts have been made towards applying virtualization concepts within Grid environment. First, Virtual Workspace (Keahey et al., 2005) represents an approach for the integration of Grid and virtualization. The approach allows users to define a virtual environment on top of a Grid environment. The basic idea is to allow Globus middleware to run on top of the Xen hypervisor. From this work, Moore et al. (2002) and Foster et al. (2006) proposed a virtual cluster that could be formed for the management of an execution environment using virtual machines. This approach also extends virtual workspace and investigates the performance issues related to the execution of applications on a virtual cluster. A middleware system, Violin (Jiang and Xu, 2004) proposed a virtual internetworking infrastructure which manages virtual machines and, virtual network technologies to create virtual distributed environments. This also enables the execution of distributed applications which require customized execution, network environments and security. The In-Vigo project (S. Adabala and Zhu. 2005) was built using components that allow for the virtualization of Grid resources and user interfaces. Additionally, the In-Vigo has a distributed virtual file system to facilitate data transfer across Grid resources; virtual machines provide isolation, resource integrity, legacy software support, environment encapsulation and customization.

Another effort has been made to incorporate virtual machine technology in the Globus Grid infrastructure (Herrera et al., 2005). This architecture deploys virtual machines across the Grid site and manages jobs using the Gridway scheduler. A solution for controlling and managing a Grid infrastructure is also described in (Montes et al., 2005) where the elements of a gLite Grid middleware can be incorporated into virtual machines, which then can be deployed throughout the Grid infrastructure as needed, reducing significantly the cost of system maintenance and management. The CARE resource broker (Somasundaram et al., 2010) also uses virtualization technology to deploy virtual machines across the Globus Grid site. The CARE broker deploys the required number of virtual machines in the Globus Grid infrastructure to meet the application requirements. CARE can create virtual clusters dynamically where each virtual machine is configured with the required software execution environment for facilitating application execution on-demand.

All these research efforts address the benefits of virtualization in a Grid environment. Our work is similar in that we also, in effect, make use of virtualization technology to provide the execution environment for running Grid jobs. However, our work differs in that:

1. We propose a novel system architecture which manages the jobs across a Grid infrastructure without the need to define additional interfaces;

2. In our implementation, there is no need to modify the existing configuration of Grid systems; and

3. Most importantly, unlike prior works, our architecture makes it possible to automatically and dynamically deploy additional Grid worker nodes using virtualization technology based on workload demands without user intervention.

C.3 Architectural Overview

This section presents an architectural overview of our system. The system relies on virtualization technology to dynamically provision additional worker nodes for pending jobs. At the high level, users may submit their jobs either to the meta-scheduler (MS) level or directly to the Local Resource Manager (LRM).

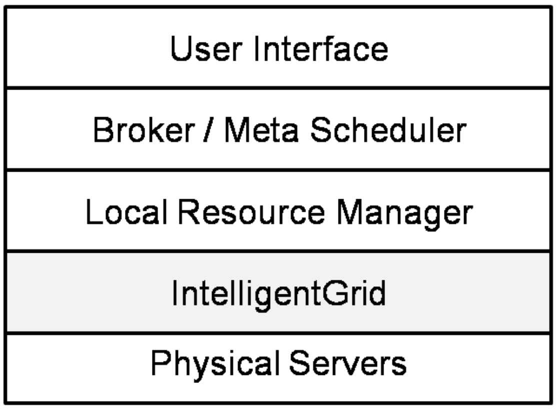


Figure C.1: High Level Architectural Overview of IntelligentGrid in relation to Grid Infrastructure.

Figure C.1 shows the architectural overview of how the IntelligentGrid is integrated with the overall Grid infrastructure. At the high level, the meta-scheduler or broker performs the necessary resource matching process for the allocation of each job to the most appropriate Resource Manager. As can be seen from the figure, the IntelligentGrid component resides between the LRM and hardware levels. All pending jobs that cannot be allocated by the MS (either due to an insufficient amount of nodes or software unavailability) are redirected to the LRM, which is supported by the IntelligentGrid, for execution.

The IntelligentGrid further comprises a number of core components as depicted in Figure C.2. The LRMS Queue Monitor is responsible for discovering any newly arrivedl jobs from the Workload Management System (WMS). The WMS has a list of all available Local Resource Managers which are known as Computing Element (CEs). Based on the job requirements, the WMS determines the most appropriate

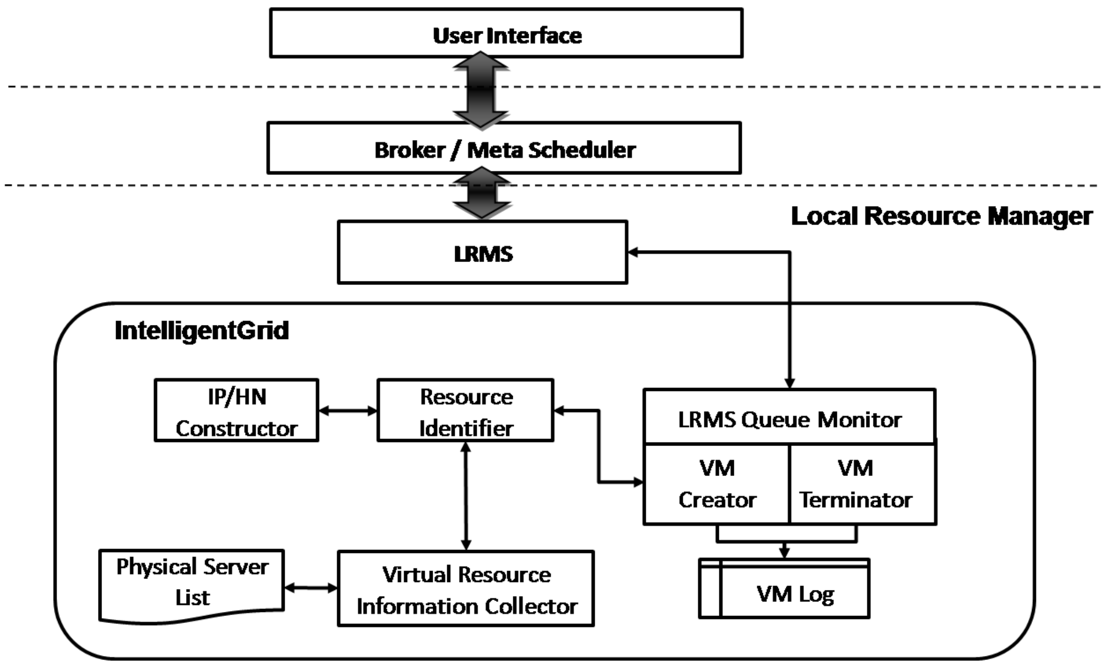


Figure C.2: Core components of IntelligentGrid and their interactions with one another.

CE to submit the job to. Subsequently, the LRMS Queue Monitor picks the highest priority job in the queue and notifies the LRM to reserve resources for this particular job. It then determines the amount of additional resource nodes needed to execute this job and submits this information to the VM Creator. Based on the amount of additional nodes required, the VM Creator communicates with the Resource Identifier to identify suitable physical servers for the creation of the required number of virtual machines (VMs). The Resource Identifier communicates with the Virtual Resource Information Collector (VRIC) to obtain hardware information such as hard disk capacity, free Random Access Memory (RAM) and storage capacity, and most importantly, the number of CPUs of all accessible physical servers that are available from the LRM. The VRIC is very similar to the LRMS Queue Monitor, but instead it periodically extracts the most recent machine information such as number of free CPUs, free RAM and storage capacity.

The Resource Identifier further consults the IP / Hostname constructor to obtain a list of suitable IP addresses and hostnames for the deployment of new VMs. Next, it requests the IP / Hostname constructor to generate a hostname and an IP address for each new VM. Subsequently, the Resource Identifier reserves the RAM and storage capacity for the Resource Identifier, and then the Resource Identifier returns the information of available physical servers to the VM Creator.

Based on the information provided, the VM Creator starts to perform the configuration necessary to deploy new Grid worker nodes on its reserved physical server(s). The configuration involves getting the appropriate VM image from the image repository and the VM image is configured to support serial and/or parallel execution of jobs. Once the VM image has been configured, the hostname and IP address of the newly configured Grid worker node is submitted to the LRM for registration. The LRM registers the newly added worker nodes to the resource pool and it schedules the reserved job with the newly configured Grid worker node(s). The LRMS Queue Monitor monitors the execution of the job and upon its completion, it notifies the VM Terminator. The VM Terminator subsequently notifies the LRM to deregister the worker nodes from the resource pool and then removes the VMs from the physical server(s) immediately.

C.4 Implementation

The IntelligentGrid prototype uses gLite middleware to construct the Grid infrastructure and Xen hypervisor for the virtualization technology. The Local Resource Manager (LRM) is based on the Open PBS batch scheduler.

The gLite Grid architecture consists of the following major components: User Interface (UI), Workload Management System (WMS), Computing Element (CE) and Worker node (WN). The User Interface (UI) resides at the top level and enables users to submit jobs. Jobs which are submitted from the UI are sent to the Workload Management System (WMS). The WMS has a list of all available Local Resource Managers which are known as Computing Element (CEs). Based on the job requirements, the WMS determines the most appropriate CE to submit the job to. If there are sufficient and suitable worker nodes (WNs) to run the job, the job is executed immediately; otherwise, the job is held in the queue until sufficient worker nodes are available.

The IntelligentGrid prototype is implemented using Java and UNIX shell scripts. The LRMS Queue Monitor periodically pulls information on queued status jobs at the CE by using the 'qstat –a' command and extracts the node requirement such as the required number of CPUs, number of nodes, and memory for each queued job. It then sends the extracted node requirement information to the VM Creator component that resides at the LRM level. Specifically, the VM Creator component extracts the following information: (1) Job ID as defined by the LRM; (2) number of requested nodes and CPUs; and (3) minimum RAM capacity. Similarly, the VM Creator queries the Resource Identifier for IP addresses and hostnames to be used for virtual machine deployment.

When all the relevant parameters are obtained, the VM Creator starts to prepare and configure the required number of virtual machines from a VM image file. A VM image file is a pre-configured image file that is comprised of the Scientific Linux version 4 OS with gLite's Worker Node service. It is also configured with the relevant MPICH library for supporting the execution of parallel jobs in Grid environments. To ensure that cross-communication can be established between existing worker nodes and newly deployed nodes, the VM Creator appends the IP addresses of newly created VMs to both the CE and the MPICH configuration files. Similarly, each time a worker node is removed upon job completion, the VM Terminator removes the corresponding IP addresses from the CE and the MPICH configuration files as appropriate. The pseudo code for VM resource provisioning is shown in 9.

C.5 Performance

Performance evaluation was conducted to answer the following questions: a) How long would it take to deploy Grid worker nodes with IntelligentGrid, i.e., taking into consideration deployment and configuration latency? Specifically, we are interested in finding out how the delays in our approach compare to

Algorithm 9 Pseudo code for VM resource provisioning.

```
1. Get high-priority, queued job $J$ from LRM queue.
2. Process $K$ jobs on available physical worker nodes.
3. Calculate the number of virtual machines N which can be created using the information about
   physical machines.
4. Get the number of jobs $J$-$K$ which can be processed by the creation of $M$ ($M$<=$N$) virtual ma-
   chines.
5. Put $P$ jobs on ‘“held”’ state and invoke the deployment process of virtual machines.
6. Monitor the VM creation process and after deployment, configure necessary settings and submit
   job.
7. Track job state and after successful execution remove virtual machines after completion of job.
8. Repeat steps from (1) until all jobs have been processed.
```

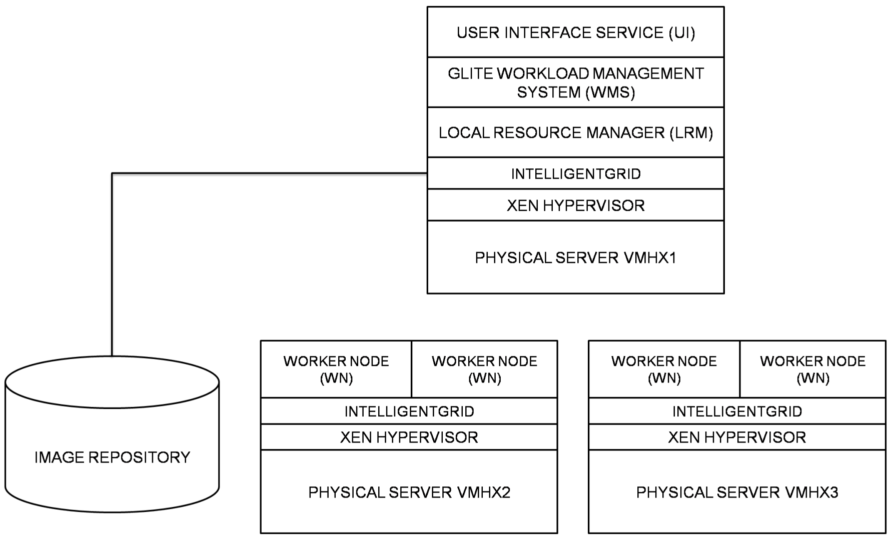


Figure C.3: Experimental testbed.

those of current Grid approaches; b) What is the impact of deployment request and task allocation costs on the practical aspect of resource provisioning strategies employed by IntelligentGrid?

First, we assume a typical gLite Grid environment where a Grid broker (meta-scheduler) is used to co-ordinate heterogeneous and geographically distributed resources from multiple LRMs. Figure C.3 shows an overview of our experimental test-bed. The test-bed comprises three physical servers namely, *vmhx1*, *vmhx2*, and *vmhx3*, each equipped with four processors with Intel(R) Xeon(R) 2.66 GHz or Intel(R) Xeon(R) CPU 2.80 GHz processors, RAM from 2 GB to 30 GB, and running OpenSUSE 11.1 with Xen hypervisor.

The first experiment is to measure (1) the amount of time it takes for a job to be served and (2) the amount of time it takes for a node to be allocated with a rented node as soon as the rental request is initiated. Timings are measured from two points: the time it takes for the LRMS Queue Monitor to

IntelligentGrid Components	Time in minutes
LRMS Queue Monitor	2.4 (0.4)
VM Creator	12.2 (0.7)
Resource Identifier	1.8 (0.3)
Total time	16.4 (1.6)

Table C.2: Breakdown of the total overhead time for IntelligentGrid with job size of 4. Small figures in parentheses are standard deviations.

identify a queued job and to initiate a resource provisioning decision as soon as the job is queued due to insufficient resources, and the time it takes for the Resource Identifier to discover physical servers for VM deployment and the VM Creator to configure a new worker node so that it is ready for allocation. We timed certain major IntelligentGrid operations, measuring the cost of individual operations, as well as the total overhead time required to make provisioning requests. In particular, we measure the invocation costs of the LRMS Queue Monitor, the Resource Identifier, and the VM Creator, as well as the VM Terminator invocation operations.

Table C.1 presents the total overhead time taken by the IntelligentGrid to deploy worker nodes with increasing job size, i.e., number of requested nodes for each job. For each experiment, we submit a dummy parallel job that runs for exactly 5 minutes. Each experiment is repeated with the same job but for different job sizes. We then measure the total time it takes for the worker nodes to be deployed for each job. As indicated in Table C.1, it would take an average of 7.4 minutes to deploy a Grid worker node, and the time it takes to provision new worker nodes increases linearly as the job size increases. For instance, it would take approximately 16.4 minutes to deploy a parallel job with a job size of 4.

Job Size	Total Overhead in minutes
1	7.4 (1.2)
2	11.3 (1.4)
3	14.5 (1.7)
4	16.4 (1.9)
8	21.8 (1.8)

Table C.1: Breakdown of the total overhead time for IntelligentGrid with job size of 4. Small figures in parentheses are standard deviations.

Table C.2 further shows the breakdown of the total overhead time for a job with size of 4. We can observe that the time overheads for queuing time and allocation time by the LRMS Queue Monitor are negligible, but the VM Creator takes a considerable time to deploy virtual machines on physical servers. However, we would argue that long-running jobs without IntelligentGrid may be queued for several hours (which is currently quite common for batch jobs). Hence, an additional delay of 7~20 minutes for scheduling is insignificant. Nonetheless, we feel that the deployment time can be improved by further optimizing the VM deployment policy, which is necessary for supporting execution of interactive jobs. We conclude that even though the VM deployment process still needs to be refined further, Intelligent-Grid is fairly scalable in terms of provisioning more than one VM simultaneously when compared to a conventional Grid setting with no deployment mechanism.

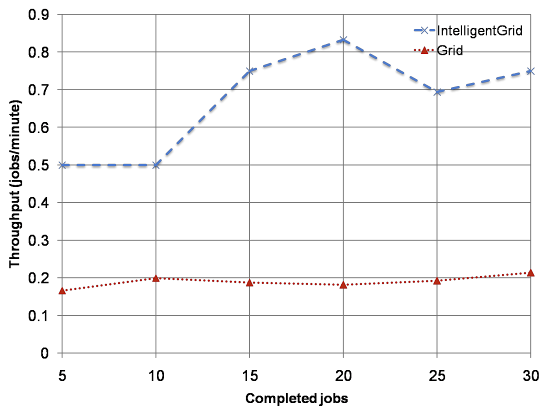


Figure C.4: Experimental performance of Intelligent vs. conventional gLite Grid system.

Next, we carried out a set of experiments to compare the performance of the IntelligentGrid in comparison to a standard Grid system. Again, the same test-bed configurations are used, which comprises three physical servers. Figure C.4 presents our results in terms of throughput. Throughput is computed as the number of completed job per minute over time, as more jobs get completed. For each experiment, the bench-marking measurements consist of the execution of 30 jobs whereby each job runs a computation for 10 minutes on average on a worker node. The results show that the IntelligentGrid system outperforms the standard Grid system. As can be seen, the IntelligentGrid achieves significantly higher throughput, an increase of up to 50% more than the standard gLite Grid system. This high throughput is due to the efficiency of the IntelligentGrid in reducing the number of pending jobs in the queue by deploying additional worker nodes. As more jobs are being served at any one time, the system is able to complete more jobs.

It is interesting to note that 17 additional worker nodes are being deployed on the three physical nodes at the end of our experiments for all 30 jobs. This has a significant impact on performance because up to 8 jobs can be executed simultaneously. Additionally, these results show that high throughput can be achieved without the need to use an additional physical server. Nonetheless, it is envisaged that we can obtain a much higher improvement in throughput by adding an increasing number of physical nodes to the LRM.

C.6 Conclusion

We have presented the IntelligentGrid system that provides the mechanisms whereby resource nodes can be automatically provisioned in a dynamic and incremental fashion based on user workloads. In particular, we have focused on the design and implementation of general, extensible abstractions of IntelligentGrid components. IntelligentGrid leverages virtual machine (VM) technology to deploy additional Grid worker nodes for the execution of serial and parallel jobs. Virtualization technology offers effective resource management mechanisms such as isolated, secure job scheduling and utilization of computing resources. Our experiment results showed the feasibility of IntelligentGrid as well as a higher throughput as compared to a typical Grid environment.

To the best of our knowledge, IntelligentGrid is the first system that presents a full-fledged dynamic resource provisioning system without the need for additional interfaces or configuration changes to existing gLite Grid systems. The system can be deployed easily without the need to reconfigure the local resource manager (LRM). The only requirement is to have the hypervisors installed on physical servers. The rest of the configuration occurs dynamically, through the interactions of various IntelligentGrid components. By simplifying configuration, IntelligentGrid makes it possible to automatically and dynamically provision additional Grid worker nodes based on workload demands without user intervention. Our future work includes improving image creation and deploying IntelligentGrid to a production environment.

Bibliography

Agrawal, Kunal et al. (2006). "Adaptive scheduling with parallelism feedback". In: *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, New York, USA: ACM, pp. 100–109. ISBN: 1-59593-189-9. DOI: `http://doi.acm.org/10.1145/1122971.1122988`.

Alfieri, R. et al. (2005). "From gridmap-file to VOMS: managing authorization in a Grid environment". In: *Future Generation Computer Systems* 21.4. High-Speed Networks and Services for Data-Intensive Grids: the DataTAG Project, pp. 549 –558. ISSN: 0167-739X. DOI: `DOI:10.1016/j.future.2004.10.006`.

Alonso, J. M. et al. (2007). "GRID technology for structural analysis". In: *Adv. Eng. Softw.* 38.11-12, pp. 738–749. ISSN: 0965-9978. DOI: `http://dx.doi.org/10.1016/j.advengsoft.2006.08.029`.

Amazon, E.C. (2009). "Amazon elastic compute cloud". In: *Retrieved Feb* 10.

Andreetto, P. et al. (2008). "The gLite workload management system". In: *Journal of Physics: Conference Series*. Vol. 119. IOP Publishing, p. 062007.

Andrews, Tony et al. (2003). *Business Process Execution Language for Web Services*. 2nd public draft release. Version 1.1. URL: `http://www.ibm.com/developerworks/webservices/library/ws-bpel/`.

Andrieux, Alain et al. (2006). *Web Services Agreement Specification (WS-Agreement)*. URL: `https://forge.gridforum.org/projects/graap-wg/`.

Andrzejak, Artur et al. (2002). *Bounding the Resource Savings of Utility Computing Models*. Tech. rep.

Assuncao, Marcos Dias de and Rajkumar Buyya (2008). "A Cost-Aware Resource Exchange Mechanism for Load Management across Grids". In: *ICPADS '08: Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*. Washington, DC, USA: IEEE Computer Society, pp. 213–220. ISBN: 978-0-7695-3434-3. DOI: `http://dx.doi.org/10.1109/ICPADS.2008.11`.

AuYoung, Alvin et al. (2006). "Service contracts and aggregate utility functions". In: *HPDC*, pp. 119–131.

Barham, Paul et al. (2003a). "Xen and the art of virtualization". In: *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. Bolton Landing, NY, USA:

ACM, pp. 164–177. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945462. URL: http://dx.doi.org/10.1145/945445.945462.

Barham, Paul et al. (2003b). “Xen and the art of virtualization”. In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. SOSP ’03. Bolton Landing, NY, USA: ACM, pp. 164–177. ISBN: 1-58113-757-5.

Barmouta, Alexander and Rajkumar Buyya (2002). “GridBank: A Grid Accounting Services Architecture (GASA) for Distributed Systems Sharing”. In: *Proceedings of the 17th Annual International Parallel and Distributed Processing Symposium (IPDPS 2003), IEEE Computer*. Society Press, pp. 22–26.

Basney, J., M. Humphrey, and V. Welch (2005). “The MyProxy online credential repository”. In: *Software: Practice and Experience* 35.9, pp. 801–816. ISSN: 1097-024X.

Basney, Jim, Miron Livny, and Paolo Mazzanti (2000). “Utilizing Widely Distributed Computational Resources Efficiently with Execution Domains”. In: *COMPUTER PHYSICS COMMUNICATIONS* 140, pp. 200–1.

Basney, Jim et al. (2005). “Negotiating Trust on the Grid”. In: *Semantic Grid: The Convergence of Technologies*. Ed. by Carole Goble, Carl Kesselman, and York Sure. Dagstuhl Seminar Proceedings 05271. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. URL: http://drops.dagstuhl.de/opus/volltexte/2005/387.

Beckman, Pete et al. (2006). “SPRUCE: A System for Supporting Urgent High-Performance Computing”. In: *IFIP WoCo9 Conference Proceedings, Arizona*.

Beguelin, Adam et al. (1991). *A User’s Guide to PVM Parallel Virtual Machine*. Tech. rep. Knoxville, TN, USA.

Benkner, Siegfried et al. (2004). “VGE - A Service-Oriented Grid Environment for On-Demand Supercomputing”. In: *GRID ’04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*. Washington, DC, USA: IEEE Computer Society, pp. 11–18. ISBN: 0-7695-2256-4. DOI: http://dx.doi.org/10.1109/GRID.2004.65.

Berlich, Ruediger et al. (2006). “EGEE: Building a Pan-European Grid Training Organisation”. In: *Fourth Australasian Symposium on Grid Computing and e-Research (AusGrid 2006)*. Ed. by Rajkumar Buyya and Tianchi Ma. Vol. 54. CRPIT. Hobart, Australia: ACS, pp. 105–111.

Bockelman, B. (2009). “Using Hadoop as a grid storage element”. In: *Journal of Physics: Conference Series*. Vol. 180. IOP Publishing, p. 012047.

Bolte, M. et al. (2010). “Non-intrusive virtualization management using libvirt”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*. IEEE, pp. 574–579.

Burge, Jennifer, Parthasarathy Ranganathan, and Janet L. Wiener (2007). “Cost-aware scheduling for heterogeneous enterprise machines (CASH’EM)”. In: *CLUSTER ’07: Proceedings of the 2007 IEEE International Conference on Cluster Computing*. Washington, DC, USA: IEEE Computer Society, pp. 481–487. ISBN: 978-1-4244-1387-4. DOI: `http://dx.doi.org/10.1109/CLUSTR.2007.4629273`.

Buyya, R., D. Abramson, and S. Venugopal (2005). "The Grid Economy". In: *Proceedings of the IEEE* 93.3, pp. 698 –714. ISSN: 0018-9219. DOI: `10.1109/JPROC.2004.842784`.

Buyya, Rajkumar (2002). "Economic-based Distributed Resource Management and Scheduling for Grid Computing". In: *CoRR* cs.DC/0204048.

Buyya, Rajkumar, David Abramson, and Jonathan Giddy (2000). "Nimrod/G: An Architecture for a Resource Management and SchedulingSystem in a Global Computational Grid". In: URL: `citeseer.ist.psu.edu/buyya00nimrodg.html`.

Buyya, Rajkumar et al. (2005). "Scheduling parameter sweep applications on global Grids: a deadline and budget constrained cost-time optimization algorithm". In: *Softw. Pract. Exper.* 35.5, pp. 491–512. ISSN: 0038-0644. DOI: `http://dx.doi.org/10.1002/spe.646`.

Calleja, Mark et al. (2005). "grid infrastructure for molecular simulations: The eMinerals minigrid as a prototype integrated compute and data grid. Molecular Simulations 31". In: *RJ Allan, C Chapman, W Emmerich, P Wilson, J Brodholt, A.Thandavan, VN Alexandrov. Collaborative* 31, pp. 303–313.

Cao, J. et al. (2003). "GridFlow: WorkFlow Management for Grid Computing". In: *THIRD IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID (CCGRID'03)*. IEEE Computer Society, pp. 198–205.

Caracas, A. and J. Altmann (2007). "A pricing information service for grid computing". In: *Proceedings of the 5th international workshop on Middleware for grid computing: held at the ACM/IFIP/USENIX 8th International Middleware Conference*. ACM, pp. 1–6.

Casanova, Henri and Jack Dongarra (1996). "NetSolve: a network server for solving computational science problems". In: *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*. Pittsburgh, Pennsylvania, United States: IEEE Computer Society, p. 40. ISBN: 0-89791-854-1. DOI: `http://doi.acm.org/10.1145/369028.369111`.

Cencerrado, Andrés, Miquel Ángel Senar, and Ana Cortés (2009). "Support for Urgent Computing Based on Resource Virtualization". In: *ICCS '09: Proceedings of the 9th International Conference on Computational Science*. Baton Rouge, LA: Springer-Verlag, pp. 227–236. ISBN: 978-3-642-01969-2.

Chakravarti, Arjav J., Gerald Baumgartner, and Mario Lauria (2004). "The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network". In: *IEEE Transactions on Systems, Man, and Cybernetics*, pp. 96–103.

Chase, Jeffrey S. et al. (2003). "Dynamic Virtual Clusters in a Grid Site Manager". In: *hpdc* 00, p. 90. ISSN: 1082-8907. DOI: `http://doi.ieeecomputersociety.org/10.1109/HPDC.2003.1210019`.

Chester, Timothy M. (2001). "Cross-Platform Integration with XML and SOAP". In: *IT Professional* 3, pp. 26–34. ISSN: 1520-9202. DOI: `http://doi.ieeecomputersociety.org/10.1109/6294.952977`.

Chhetri, Mohan Baruwal et al. (2006). “A Coordinated Architecture for the Agent-based Service Level Agreement Negotiation ofWeb Service Composition”. In: *Software Engineering Conference, Australian* 0, pp. 90–99. ISSN: 1530-0803. DOI: `http://doi.ieeecomputersociety.org/10.1109/ASWEC.2006.1`.

Chien, Andrew et al. (1997). “High performance virtual machines (HPVM): Clusters with supercomputing APIs and performance”. In: *In Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*.

Choi, Hyung-Jun et al. (2009). “A super-metascheduler-based approach for integrating multiple heterogeneous grids”. In: *ICACT’09: Proceedings of the 11th international conference on Advanced Communication Technology*. Gangwon-Do, South Korea: IEEE Press, pp. 2065–2070. ISBN: 978-8-9551-9138-7.

Christensen, Erik et al. (2001). *Web Service Definition Language (WSDL)*. Tech. rep. URL: `http://www.w3.org/TR/wsdl`.

Chun, B. and D. Culler (2002a). *User-centric Performance Analysis of Market-based Cluster Batch Schedulers*. URL: `citeseer.ist.psu.edu/chun02usercentric.html`.

Chun, B.N. et al. (2005). “Mirage: a microeconomic resource allocation system for sensornet testbeds”. In: *Embedded Networked Sensors, IEEE Workshop on* 0, pp. 19–28. DOI: `http://doi.ieeecomputersociety.org/10.1109/EMNETS.2005.1469095`.

Chun, Brent N. and David E. Culler (2000). “REXEC: A Decentralized, Secure Remote Execution Environment for Clusters”. In: *CANPC ’00: Proceedings of the 4th International Workshop on Network-Based Parallel Computing*. London, UK: Springer-Verlag, pp. 1–14. ISBN: 3-540-67879-4.

— (2002b). “User-Centric Performance Analysis of Market-Based Cluster Batch Schedulers”. In: *Cluster Computing and the Grid, IEEE International Symposium on* 0, p. 30. DOI: `http://doi.ieeecomputersociety.org/10.1109/CCGRID.2002.1017109`.

Chunlin, Li and Li Layuan (2005). “Pricing and resource allocation in computational grid with utility functions”. In: *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on*. Vol. 2, 175 –180 Vol. 2. DOI: `10.1109/ITCC.2005.231`.

Clark, K.P. et al. (2010). “Secure Monitoring of Service Level Agreements”. In: *Availability, Reliability and Security, International Conference on* 0, pp. 454–461. DOI: `http://doi.ieeecomputersociety.org/10.1109/ARES.2010.33`.

Codispoti, G et al. (2010). “Use of the gLite-WMS in CMS for production and analysis”. In: *Journal of Physics: Conference Series* 219.6, p. 062007. URL: `http://stacks.iop.org/1742-6596/219/i=6/a=062007`.

Community Scheduler Framework (CSF) (2010). GT 4.20. URL: `www.globus.org/gridsoftware/computation/csf.php`.

Costa, P. et al. (2009). “Autonomous Resource Selection for Decentralized Utility Computing”. In: pp. 561 –570. DOI: `10.1109/ICDCS.2009.70`.

Coveney, P.V. et al. (2010). “Large scale computational science on federated international grids: The role of switched optical networks”. In: *Future Generation Computer Systems* 26.1, pp. 99 –110. ISSN: 0167-739X. DOI: DOI:10.1016/j.future.2008.09.013. URL: http://www.sciencedirect.com/science/article/B6V06-4TN82BY-1/2/7e31c36d85f4bebb4a0ddffcd3695198.

Culler, David E. (1997). “Parallel computing on the berkeley now”. In: *In Proceedings of the 9th Joint Symposium on Parallel Processing (JSPP 97*.

Czajkowski, K. et al. (2001). *Grid Information Services for Distributed Resource Sharing*. URL: http://citeseer.ist.psu.edu/czajkowski01grid.html.

Czajkowski, K. et al. (2002). *SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems*. URL: citeseer.ist.psu.edu/czajkowski02snap.html.

Czajkowski, Karl, Ian T. Foster, and Carl Kesselman (1999). “Resource Co-Allocation in Computational Grids”. In: *HPDC*. URL: citeseer.ist.psu.edu/czajkowski99resource.html.

Das, A. and D. Grosu (2005a). “Combinatorial auction-based protocols for resource allocation in grids”. In: ISSN: 1530-2075.

Das, Anubhav and Daniel Grosu (2005b). “Combinatorial Auction-Based Protocols for Resource Allocation in Grids”. In: *IPDPS ’05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS’05) - Workshop 13*. Washington, DC, USA: IEEE Computer Society, p. 251.1. ISBN: 0-7695-2312-9. DOI: http://dx.doi.org/10.1109/IPDPS.2005.140.

Deelman, Ewa et al. (2005). “Pegasus: A framework for mapping complex scientific workflows onto distributed systems”. In: *Scientific Programming* 13.3, pp. 219–237.

Deelman, Ewa et al. (2008). “The cost of doing science on the cloud: the Montage example”. In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. SC ’08. Austin, Texas: IEEE Press, 50:1–50:12. ISBN: 978-1-4244-2835-9. URL: http://portal.acm.org/citation.cfm?id=1413370.1413421.

Deelman, Ewa et al. (2009). “Workflows and e-Science: An overview of workflow system features and capabilities”. In: *Future Gener. Comput. Syst.* 25 (5), pp. 528–540. ISSN: 0167-739X. DOI: 10.1016/j.future.2008.06.012. URL: http://portal.acm.org/citation.cfm?id=1507767.1507921.

D.G.Feitelson (2009). *Logs of Real Parallel Workloads from Production Systems*. URL: http://www.cs.huji.ac.il/labs/parallel/workload/logs.html.

Di Caro, Gianni and Marco Dorigo (1998). “AntNet: Distributed Stigmergetic Control for Communications Networks”. In: *Journal of Artificial Intelligence Research* 9, pp. 317–365. URL: http://citeseer.ist.psu.edu/dicaro98antnet.html.

Drost, Niels, Rob V. van Nieuwpoort, and Henri Bal (2006). “Simple Locality-Aware Co-allocation in Peer-to-Peer Supercomputing”. In: *CCGRID ’06: Proceedings of the Sixth IEEE International*

Symposium on Cluster Computing and the Grid. Washington, DC, USA: IEEE Computer Society, p. 14. ISBN: 0-7695-2585-7.

Duan, Zhenhai, Zhi-Li Zhang, and Yiwei Thomas Hou (2003). “Service overlay networks: SLAs, QoS, and bandwidth provisioning”. In: *IEEE/ACM Trans. Netw.* 11.6, pp. 870–883. ISSN: 1063-6692. DOI: `http://dx.doi.org/10.1109/TNET.2003.820436`.

Dube, Nicolas and Marc Parizeau (2008). “Utility Computing and Market-Based Scheduling: Shortcomings for Grid Resources Sharing and the Next Steps”. In: *High Performance Computing Systems and Applications, Annual International Symposium on* 0, pp. 59–68. ISSN: 1550-5243. DOI: `http://doi.ieeecomputersociety.org/10.1109/HPCS.2008.29`.

Eberhart, Russell C., Yuhui Shi, and James Kennedy (2001). *Swarm Intelligence (The Morgan Kaufmann Series in Evolutionary Computation)*. 1st. Morgan Kaufmann. ISBN: 1558605959. URL: `http://www.worldcat.org/isbn/1558605959`.

EGEE Project (2007). *Lightweight Middleware for Grid Computing*. `http://glite.web.cern.ch/glite/`.

Eickermann, Thomas et al. (2007). *Co-allocation of MPI Jobs with the VIOLA Grid MetaScheduling Framework*.

Elmroth, Erik and Johan Tordsson (2009). “A standards-based Grid resource brokering service supporting advance reservations, coallocation, and cross-Grid interoperability”. In: *Concurrency and Computation: Practice and Experience* 21.18, pp. 2298–2335.

Engelbrecht, Gerhard and Siegfried Benkner (2009). “A Service-Oriented Grid Environment with On-demand QoS Support”. In: *Proceedings of the 2009 Congress on Services - I*. Washington, DC, USA: IEEE Computer Society, pp. 147–150. ISBN: 978-0-7695-3708-5. DOI: `10.1109/SERVICES-I.2009.84`. URL: `http://portal.acm.org/citation.cfm?id=1590963.1591536`.

Environments, G. and W. Group. *Global Grid Forum*. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.5270`.

Erl, Thomas (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR. ISBN: 0131858580.

Ernemann, C. et al. (2002). “On advantages of grid computing for parallel job scheduling”. In: *ccgrid*. Published by the IEEE Computer Society, p. 39.

Erwin, Dietmar (2001). “UNICORE - A Grid Computing Environment”. In: *Lecture Notes in Computer Science*. Springer-Verlag, pp. 825–834.

Fahringer, T., J. Qin, and S. Hainzer (2005). “Specification of grid workflow applications with AGWL: an Abstract Grid Workflow Language”. In: *Cluster Computing and the Grid, IEEE International Symposium on* 2, pp. 676–685. DOI: `http://doi.ieeecomputersociety.org/10.1109/CCGRID.2005.1558629`.

Fahringer, Thomas et al. (2005). "ASKALON: a tool set for cluster and Grid computing: Research Articles". In: *Concurr. Comput. : Pract. Exper.* 17.2-4, pp. 143–169. ISSN: 1532-0626. DOI: `http://dx.doi.org/10.1002/cpe.v17:2/4`.

Fakhouri, S. et al. (2001). "Oceano - SLA Based Management of a Computing Utility". In: *In Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, pp. 855–868.

Feitelson, D. et al. (1997). "Theory and practice in parallel job scheduling". In: *Job Scheduling Strategies for Parallel Processing*. Springer, pp. 1–34.

Feitelson, D. G. (2005). "The supercomputer industry in light of the Top500 data". In: *Computing in Science & Engineering [see also IEEE Computational Science and Engineering]* 7.1, pp. 42–47. URL: `http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=1377075`.

Feitelson, Dror G. and Larry Rudolph (1995). "Parallel Job Scheduling: Issues and Approaches". In: *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. London, UK: Springer-Verlag, pp. 1–18. ISBN: 3-540-60153-8.

F.Heine et al. (2005). "On the Impact of Reservations from the Grid on Planning-Based Resource Management". In: *International Workshop on Grid Computing Security and Resource Management*. Atlanta, USA, pp. 155–162.

Field, L and M Schulz (2008). "Grid interoperability: the interoperations cookbook". In: *Journal of Physics: Conference Series* 119.1, p. 012001. URL: `http://stacks.iop.org/1742-6596/119/i=1/a=012001`.

Fiscato, Marco, Paolo Costa, and Guillaume Pierre (2008). "On the Feasibility of Decentralized Grid Scheduling". In: *SASOW '08: Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. Washington, DC, USA: IEEE Computer Society, pp. 225–229. ISBN: 978-0-7695-3553-1. DOI: `http://dx.doi.org/10.1109/SASOW.2008.64`.

Foster, I., D. Gannon (eds.), and D. Gannon Indiana U (2003). *The Open Grid Services Architecture Platform*.

Foster, I. and C. Kesselman (1997). "Globus: A Metacomputing Infrastructure Toolkit". In: *The International Journal of Supercomputer Applications and High Performance Computing* 11.2, pp. 115–128. URL: `citeseer.ist.psu.edu/foster96globus.html`.

Foster, I. et al. (1999a). "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation". In: *Proceedings of the International Workshop on Quality of Service*. URL: `citeseer.ist.psu.edu/foster99distributed.html`.

Foster, I. et al. (2005). "Modeling and managing state in distributed systems: The role of OGSI and WSRF". In: *Proceedings of the IEEE* 93.3, pp. 604–612. ISSN: 0018-9219.

Foster, Ian, Carl Kesselman, and Steven Tuecke (2001). "The Anatomy of the Grid: Enabling Scalable Virtual Organizations". In: *Int. J. High Perform. Comput. Appl.* 15.3, pp. 200–222. ISSN: 1094-3420. DOI: `http://dx.doi.org/10.1177/109434200101500302`.

Foster, Ian et al. (1998). “A security architecture for computational grids”. In: *CCS ’98: Proceedings of the 5th ACM conference on Computer and communications security*. San Francisco, California, United States: ACM, pp. 83–92. ISBN: 1-58113-007-4. DOI: http://doi.acm.org/10.1145/288090.288111.

Foster, Ian et al. (1999b). “A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation”. In: *IN PROCEEDINGS OF THE INTERNATIONAL WORKSHOP ON QUALITY OF SERVICE*, pp. 27–36.

Foster, Ian et al. (2002). “The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration”. In:

Foster, Ian T. et al. (2006). “Virtual Clusters for Grid Communities.” In: *CCGRID*. IEEE Computer Society, pp. 513–520.

Frey, James et al. (2002). “Condor-G: A Computation Management Agent for Multi-Institutional Grids”. In: *Cluster Computing* 5, pp. 237–246.

Fu, Yun et al. (2003). “SHARP: an architecture for secure resource peering”. In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. Bolton Landing, NY, USA: ACM Press, pp. 133–148. ISBN: 1-58113-757-5. DOI: http://doi.acm.org/10.1145/945445.945459.

Gentzsch, W. (2001a). “Sun Grid Engine: Towards Creating a Compute Power Grid”. In: *CCGRID ’01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, p. 35. ISBN: 0-7695-1010-8.

— (2001b). “Sun Grid Engine: Towards Creating a Compute Power Grid”. In: *CCGRID ’01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, p. 35. ISBN: 0-7695-1010-8.

Graupner, Sven, Vadim Kotov, and Holger Trinks (2002). “Resource-Sharing and Service Deployment in Virtual Data Centers”. In: *icdcsw* 00, p. 666. DOI: http://doi.ieeecomputersociety.org/10.1109/ICDCSW.2002.1030845.

Green, Les et al. (2007). “Design of a dynamic SLA negotiation protocol for grids”. In: *GridNets ’07: Proceedings of the first international conference on Networks for grid applications*. Lyon, France: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 1–8. ISBN: 978-963-9799-02-8.

Greenberg, Albert et al. (2008). “The cost of a cloud: research problems in data center networks”. In: *SIGCOMM Comput. Commun. Rev.* 39 (1), pp. 68–73. ISSN: 0146-4833. DOI: http://doi.acm.org/10.1145/1496091.1496103. URL: http://doi.acm.org/10.1145/1496091.1496103.

Grimshaw, Andrew S., Wm. A. Wulf, and CORPORATE The Legion Team (1997). “The Legion vision of a worldwide virtual computer”. In: *Commun. ACM* 40.1, pp. 39–45. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/242857.242867.

Gropp, William, Rajeev Thakur, and Ewing Lusk (1999). *Using MPI-2: Advanced Features of the Message Passing Interface*. Cambridge, MA, USA: MIT Press. ISBN: 026257134X.

Hauswirth, M. and R. Schmidt (2005). "An overlay network for resource discovery in grids". In: *Proceedings of Sixteenth Workshop on Database and Expert Systems Applications, 2005*, pp. 343–348. URL: `http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=1508296`.

He, Qiang et al. (2009). "Lifetime service level agreement management with autonomous agents for services provision". In: *Information Sciences* 179.15. Including Special Issue on Computer-Supported Cooperative Work - Techniques and Applications, The 11th Edition of the International Conference on CSCW in Design, pp. 2591 –2605. ISSN: 0020-0255. DOI: `DOI:10.1016/j.ins.2009.01.037`. URL: `http://www.sciencedirect.com/science/article/B6V0C-4VK6NDY-2/2/e82d3416f2b35db31f4bc29800092abb`.

Henderson, Robert L. (1995). "Job Scheduling Under the Portable Batch System". In: *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. London, UK: Springer-Verlag, pp. 279–294. ISBN: 3-540-60153-8.

Herrera, J. et al. (2005). "Porting of scientific applications to Grid Computing on GridWay". In: *Sci. Program.* 13.4, pp. 317–331. ISSN: 1058-9244.

Hu, Jia, Ning Zhong, and Yong Shi (2006). "Developing Grid-based E-finance Portals for Intelligent Risk Management and Decision Making". In: *Proceeding of the 2006 conference on Advances in Intelligent IT*. Amsterdam, The Netherlands, The Netherlands: IOS Press, pp. 58–66. ISBN: 1-58603-615-7.

Huang, Ye et al. (2008). "SmartGRID: A Fully Decentralized Grid Scheduling Framework Supported by Swarm Intelligence". In: *GCC '08: Proceedings of the 2008 Seventh International Conference on Grid and Cooperative Computing*. Washington, DC, USA: IEEE Computer Society, pp. 160–168. ISBN: 978-0-7695-3449-7. DOI: `http://dx.doi.org/10.1109/GCC.2008.24`.

Hudert, Sebastian, Heiko Ludwig, and Guido Wirtz (2009). "Negotiating SLAs-An Approach for a Generic Negotiation Framework for WS-Agreement". In: *J. Grid Comput.* 7.2, pp. 225–246.

Inc, Amazon (2008). *Amazon Elastic Compute Cloud (Amazon EC2)*. Amazon Inc. URL: `http://aws.amazon.com/ec2/\#pricing`.

Iordache, George V. et al. (2007a). "A decentralized strategy for genetic scheduling in heterogeneous environments". In: *Multiagent Grid Syst.* 3.4, pp. 355–367. ISSN: 1574-1702.

— (2007b). "A decentralized strategy for genetic scheduling in heterogeneous environments". In: *Multiagent Grid Syst.* 3.4, pp. 355–367. ISSN: 1574-1702.

Iosup, Alexandru et al. (2006). "How are Real Grids Used? The Analysis of Four Grid Traces and Its Implications". In: *GRID*, pp. 262–269.

Irwin, David et al. (2005). "Self-recharging virtual currency". In: *P2PECON '05: Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*. Philadelphia, Pennsylvania, USA: ACM, pp. 93–98. ISBN: 1-59593-026-4. DOI: `http://doi.acm.org/10.1145/1080192.1080194`.

Irwin, David et al. (2006). “Sharing Networked Resources with Brokered Leases”. In: pp. 199–212. URL: `http://www.usenix.org/events/usenix06/tech/irwin.html`.

Irwin, David E., Laura E. Grit, and Jeffrey S. Chase (2004). “Balancing Risk and Reward in a Market-Based Task Service”. In: *HPDC ’04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*. Washington, DC, USA: IEEE Computer Society, pp. 160–169. ISBN: 0-7803-2175-4. DOI: `http://dx.doi.org/10.1109/HPDC.2004.5`.

Islam, M., G. Khanna, and P. Sadayappan (2008). “Revenue Maximization in Market-based Parallel Job Schedulers”. In:

Islam, M. et al. (2007). “Analyzing and Minimizing the Impact of Opportunity Cost in QoS-aware Job Scheduling”. In: *ICPP ’07: Proceedings of the 2007 International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, p. 42. ISBN: 0-7695-2933-X. DOI: `http://dx.doi.org/10.1109/ICPP.2007.16`.

Islam, Mohammad et al. (2003). “QoPS: A QoS based scheme for Parallel Job Scheduling”. In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Lect. Notes Comput. Sci. vol. 2862. Springer Verlag, pp. 252–268.

Jabri, Mohamed Amin and Satoshi Matsuoka (2010). “Authorization within grid-computing using certificateless identity-based proxy signature”. In: *HPDC ’10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. Chicago, Illinois: ACM, pp. 292–295. ISBN: 978-1-60558-942-8. DOI: `http://doi.acm.org/10.1145/1851476.1851513`.

Jackson, David B., Quinn Snell, and Mark J. Clement (2001). “Core Algorithms of the Maui Scheduler”. In: *JSSPP ’01: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*. London, UK: Springer-Verlag, pp. 87–102. ISBN: 3-540-42817-8.

Jain, Ravi et al. (1997). “Heuristics for Scheduling I/O Operations”. In: *IEEE Transactions on Parallel and Distributed Systems* 8, pp. 310–320. ISSN: 1045-9219. DOI: `http://doi.ieeecomputersociety.org/10.1109/71.584096`.

Jennings, N. R. et al. (2001). “Automated negotiation: prospects, methods and challenges”. In: *Intern. J. of Group Decision and Negotiation* 10.2, pp. 199–215.

Jiang, Xuxian and Dongyan Xu (2004). “VIOLIN: Virtual Internetworking on Overlay Infrastructure”. In: *Parallel and Distributed Processing and Applications: Second International Symposium*. Ed. by Jiannong Cao et al. Springer Berlin / Heidelberg.

John, Steven Newhouse et al. (2003). *Trading Grid Services Within the UK e-Science Grid*.

Kaushik, Neena R., Silvia M. Figueira, and Stephen A. Chiappari (2006). “Flexible Time-Windows for Advance Reservation Scheduling”. In: *MASCOTS ’06: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*. Washington, DC, USA: IEEE Computer Society, pp. 218–225. ISBN: 0-7695-2573-3. DOI: `http://dx.doi.org/10.1109/MASCOTS.2006.25`.

Keahey, K. et al. (2005). “Virtual workspaces: Achieving quality of service and quality of life in the grid”. In: *Scientific Programming* 13.4, pp. 265–275. ISSN: 1058-9244.

Keller, Alexander and Heiko Ludwig (2003). “The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services”. In: *J. Netw. Syst. Manage.* 11.1, pp. 57–81. ISSN: 1064-7570. DOI: `http://dx.doi.org/10.1023/A:1022445108617`.

Kivity, Avi (2007). “kvm: the Linux virtual machine monitor”. In: *OLS ’07: The 2007 Ottawa Linux Symposium*, pp. 225–230.

Kleban, Stephen D. and Scott H. Clearwater (2004). “Computation-at-Risk: Assessing Job Portfolio Management Risk on Clusters”. In: *ipdps* 15, 254b. DOI: `http://doi.ieeecomputersociety.org/10.1109/IPDPS.2004.1303319`.

Kobler, Rene et al. (2007). “Interactive molecular dynamics simulations on the grid”. In: *EUROCAST’07: Proceedings of the 11th international conference on Computer aided systems theory*. Las Palmas de Gran Canaria, Spain: Springer-Verlag, pp. 443–447. ISBN: 3-540-75866-6, 978-3-540-75866-2.

Kola, George et al. (2004). “Disc: A system for distributed data intensive scientific computing”. In: *In Proceedings of the 1st WORLDS*.

Kondo, D. et al. (2009). “Cost-benefit analysis of Cloud Computing versus desktop grids”. In: *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1 –12. DOI: `10.1109/IPDPS.2009.5160911`.

Koomey, Jonathan (2007). “A Simple Model for Determining True Total Cost of Ownership Data Centers”. In:

Kozuch, Michael A. et al. (2009). “Tashi: location-aware cluster management”. In: *ACDC ’09: Proceedings of the 1st workshop on Automated control for datacenters and clouds*. Barcelona, Spain: ACM, pp. 43–48. ISBN: 978-1-60558-585-7. DOI: `http://doi.acm.org/10.1145/1555271.1555282`.

Lai, Kevin (2005). “Markets are dead, long live markets”. In: *SIGecom Exch.* 5.4, pp. 1–10. DOI: `http://doi.acm.org/10.1145/1120717.1120719`.

Lai, Kevin et al. (2004). *Tycoon: an Implemention of a Distributed Market-BasedResource Allocation System*. Tech. rep. arXiv:cs.DC/0412038. Palo Alto, CA, USA: HP Labs.

Lee, Cynthia B. and Allan E. Snavely (2007). “Precise and realistic utility functions for user-centric performance analysis of schedulers”. In: *HPDC ’07: Proceedings of the 16th international symposium on High performance distributed computing*. Monterey, California, USA: ACM, pp. 107–116. ISBN: 978-1-59593-673-8. DOI: `http://doi.acm.org/10.1145/1272366.1272381`.

Lee, William, A. Stephen Mcgough, and John Darlington (2005). “Performance evaluation of the GridSAM job submission and monitoring system”. In: *In UK e-Science All Hands Meeting*, pp. 915–922.

Leff, Avraham, James T. Rayfield, and Daniel M. Dias (2003). “Service-Level Agreements and Commercial Grids”. In: *IEEE Internet Computing* 7.4, pp. 44–50. ISSN: 1089-7801. DOI: `http://dx.doi.org/10.1109/MIC.2003.1215659`.

Levy, H.M. (1984). *Capability-based computer systems*. Vol. 1984. Digital Press.

Lingrand, Diane, Tristan Glatard, and Johan Montagnat (2009). “Modeling the latency on production grids with respect to the execution context”. In: *Parallel Comput.* 35.10-11, pp. 493–511. ISSN: 0167-8191. DOI: `http://dx.doi.org/10.1016/j.parco.2009.07.003`.

Litke, Antonios et al. (2008). “Managing service level agreement contracts in OGSA-based Grids”. In: *Future Gener. Comput. Syst.* 24.4, pp. 245–258. ISSN: 0167-739X. DOI: `http://dx.doi.org/10.1016/j.future.2007.06.004`.

Liu, Hao, Amril Nazir, and Søren-Aksel Sørensen (2009). “A Software Framework to Support Adaptive Applications in Distributed/Parallel Computing”. In: *HPCC*, pp. 563–570.

Liu, Hao, Amril Nazir, and Soren-Aksel Sorenson (2007). “Resource Management Support for Smooth Application Execution in a Dynamic and Competitive Environment”. In: *Semantics, Knowledge and Grid, International Conference on* 0, pp. 438–441. DOI: `http://doi.ieeecomputersociety.org/10.1109/SKG.2007.103`.

Liu, Zhen, Mark S. Squillante, and Joel L. Wolf (2001). “On maximizing service-level-agreement profits”. In: *EC ’01: Proceedings of the 3rd ACM conference on Electronic Commerce*. Tampa, Florida, USA: ACM, pp. 213–223. ISBN: 1-58113-387-1. DOI: `http://doi.acm.org/10.1145/501158.501185`.

Livny, M. et al. (1997). “Mechanisms for high throughput computing”. In: *SPEEDUP journal* 11.1, pp. 36–40.

MacLaren, Jon (2007). “HARC: The Highly-Available Resource Co-allocator”. In: *OTM Conferences (2)*, pp. 1385–1402.

Martinaitis, Paul N., Craig J. Patten, and Andrew L. Wendelborn (2009). “Component-based stream processing "in the cloud"”. In: *Proceedings of the 2009 Workshop on Component-Based High Performance Computing*. CBHPC ’09. Portland, Oregon: ACM, 16:1–16:12. ISBN: 978-1-60558-718-9. DOI: `http://doi.acm.org/10.1145/1687774.1687790`. URL: `http://doi.acm.org/10.1145/1687774.1687790`.

Massie, M.L., B.N. Chun, and D.E. Culler (2004). “The ganglia distributed monitoring system: design, implementation, and experience”. In: *Parallel Computing* 30.7, pp. 817–840. ISSN: 0167-8191.

Michlmayr, Elke (2006). “Ant Algorithms for Search in Unstructured Peer-to-Peer Networks”. In: *ICDEW ’06: Proceedings of the 22nd International Conference on Data Engineering Workshops*. Washington, DC, USA: IEEE Computer Society, p. 142. ISBN: 0-7695-2571-7. DOI: `http://dx.doi.org/10.1109/ICDEW.2006.29`.

Moaddeli, H. R., Gh. Dastghaibyfard, and M. R. Moosavi (2008). “Flexible Advance Reservation Impact on Backfilling Scheduling Strategies”. In: *Grid and Cooperative Computing, International Conference on* 0, pp. 151–159. DOI: `http://doi.ieeecomputersociety.org/10.1109/GCC.2008.85`.

Montes, M.C. et al. (2005). “Management of a grid infrastructure in GLITE with Virtualization”. In: *Scientific Programming* 13.4, pp. 265–275. ISSN: 1058-9244.

Moore, J. et al. (2002). "Managing mixed-use clusters with Cluster-on-Demand". In: *Duke University Department of Computer Science Technical Report*.

Moreira, Jos E. et al. (1997). "Dynamic Resource Management on Distributed Systems Using Reconfigurable Applications". In: *IBM Journal of Research and Development* 41, pp. 303–330.

Mu'alem, Ahuva W. and Dror G. Feitelson (2001). "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling". In: *IEEE Trans. Parallel Distrib. Syst.* 12.6, pp. 529–543. ISSN: 1045-9219. DOI: http://dx.doi.org/10.1109/71.932708.

Napper, Jeffrey and Paolo Bientinesi (2009). "Can cloud computing reach the top500?" In: *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*. UCHPC-MAW '09. Ischia, Italy: ACM, pp. 17–20. ISBN: 978-1-60558-557-4. DOI: http://doi.acm.org/10.1145/1531666.1531671. URL: http://doi.acm.org/10.1145/1531666.1531671.

Nasser, B. et al. (2005). "Access Control Model for Inter-organizational Grid Virtual Organizations". In: pp. 537–551. DOI: 10.1007/11575863_73. URL: http://dx.doi.org/10.1007/11575863_73.

Nicole, Denis A (2005). "UNICORE and GRIP: Experiences of Grid Middleware Development". In: *In 2005 International Conference on Grid Computing and Applications*. Las Vegas, Nevada, USA, pp. 11–17.

Nivet, Patrick (2006). *A computational grid devoted to fluid mechanics*. Research Report RT-0318. INRIA, p. 28. URL: http://hal.inria.fr/inria-00069860/PDF/RT-0318.pdf.

Nurmi, Daniel et al. (2008). "The Eucalyptus Open-source Cloud-computing System". In: *Proceedings of Cloud Computing and Its Applications*. Chicago, Illinois. URL: http://eucalyptus.cs.ucsb.edu/wiki/Presentations.

Opitz, Alek, Hartmut König, and Sebastian Szamlewska (2008). "What Does Grid Computing Cost?" In: *J. Grid Comput.* 6.4, pp. 385–397.

Pal, R. and P. Hui (2011). "On the Economics of Cloud Markets". In: *Arxiv preprint arXiv:1103.0045*.

Palankar, Mayur R. et al. (2008). "Amazon S3 for science grids: a viable solution?" In: *Proceedings of the 2008 international workshop on Data-aware distributed computing*. DADC '08. Boston, MA, USA: ACM, pp. 55–64. ISBN: 978-1-60558-154-5. DOI: http://doi.acm.org/10.1145/1383519.1383526. URL: http://doi.acm.org/10.1145/1383519.1383526.

Palmieri, Francesco (2009). "Network-aware scheduling for real-time execution support in data-intensive optical Grids". In: *Future Generation Computer Systems* 25.7, pp. 794 –803. ISSN: 0167-739X. DOI: DOI:10.1016/j.future.2008.11.006. URL: http://www.sciencedirect.com/science/article/B6V06-4V1KMJ9-1/2/b5351ed0bee952592c437e6005166396.

Pan, Zhengxiang et al. (Nov. 27, 2007). "Hawkeye: A Practical Large Scale Demonstration of Semantic Web Integration." In: *OTM Workshops (2)*. Ed. by Robert Meersman, Zahir Tari, and Pilar Herrero.

Vol. 4806. Lecture Notes in Computer Science. Springer, pp. 1115–1124. ISBN: 978-3-540-76889-0.

Park, Sang-Min and Marty Humphrey (2008). "Feedback-controlled resource sharing for predictable eScience". In: *SC Conference* 0, pp. 1–11. DOI: `http://doi.ieeecomputersociety.org/10.1145/1413370.1413384`.

Peterson, Larry et al. (2006). *PlanetLab Architecture: An Overview*. Tech. rep. PDN–06–031. PlanetLab Consortium.

Petrini, Fabrizio et al. (2002). "The Quadrics Network: High-Performance Clustering Technology". In: *IEEE Micro* 22.1, pp. 46–57. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.6806`.

Petrone, Mario and Roberto Zarrelli (2006). "Enabling PVM to Build Parallel Multidomain Virtual Machines". In: *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on* 0, pp. 187–194. ISSN: 1066-6192. DOI: `http://doi.ieeecomputersociety.org/10.1109/PDP.2006.33`.

Pichot, Antoine et al. (2009). "Towards Dynamic Service Level Agreement Negotiation:An Approach Based on WS-Agreement". In: *Web Information Systems and Technologies*. Ed. by Wil Aalst et al. Vol. 18. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, pp. 107–119. ISBN: 978-3-642-01344-7.

Podhorszki, Norbert and Peter Kacsuk (2003). "Monitoring Message Passing Applications in the Grid with GRM and RGMA". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI*, pp. 603–610.

Popovici, Florentina I. and John Wilkes (2005). "Profitable services in an uncertain world". In: *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, p. 36. ISBN: 1-59593-061-2. DOI: `http://dx.doi.org/10.1109/SC.2005.58`.

Qiang He Jun Yan, Kowalczyk R. Hai Jin Yun Yang (2006). "An Agent-based Framework for Service Level Agreement Management". In: *Computer Supported Cooperative work in Design Conference, Australian* 0, pp. 90–99. ISSN: 1530-0803. DOI: `http://doi.ieeecomputersociety.org/10.1109/ASWEC.2006.1`.

Ramamritham, K., J. A. Stankovic, and P. F. Shiah (1990). "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems". In: *IEEE Trans. Parallel Distrib. Syst.* 1.2, pp. 184–194. ISSN: 1045-9219. DOI: `http://dx.doi.org/10.1109/71.80146`.

Ranganathan, Kavitha and Ian Foster (2003). "Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids". In: *Journal of Grid Computing* 1 (1). 10.1023/A:1024035627870, pp. 53–62. ISSN: 1570-7873. URL: `http://dx.doi.org/10.1023/A:1024035627870`.

Ranjan, R. and R. Buyya (2010). "Decentralized overlay for federation of Enterprise Clouds". In: *Handbook of Research on Scalable Computing Technologies*, p. 191.

Ranjan, Rajiv et al. (2007). "Decentralised Resource Discovery Service for Large Scale Federated Grids". In: *eScience*, pp. 379–387.

Rappa, M. A. (2004). "The utility business model and the future of computing services". In: *IBM Syst. J.* 43.1, pp. 32–42. ISSN: 0018-8670.

Ridge, D. et al. (1997). *Beowulf: Harnessing the power of parallelism in a pile-of-pcs*. URL: `citeseer.ist.psu.edu/ridge97beowulf.html`.

Rubach, Pawel and Michael Sobolewski (2009). "Dynamic SLA Negotiation in Autonomic Federated Environments". In: *OTM '09: Proceedings of the Confederated International Workshops and Posters on On the Move to Meaningful Internet Systems: ADI, CAMS, EI2N, ISDE, IWSSA, MONET, OnToContent, ODIS, ORM, OTM Academy, SWWS, SEMELS, Beyond SAWSDL, and COMBEK 2009*. Vilamoura, Portugal: Springer-Verlag, pp. 248–258. ISBN: 978-3-642-05289-7.

Ruth, P., P. McGachey, and D. Xu (2005). "VioCluster: Virtualization for Dynamic Computational Domains". In: *IEEE International Conference on Cluster Computing (Cluster'05)*. Vol. 0. URL: `http://www.cs.purdue.edu/homes/dxu/pubs/Cluster2005.pdf`.

S. Adabala V. Chadha, P. Chawla R. Figueiredo J. Fortes I. Krsul A. Matsunaga M. Tsugawa J. Zhang M. Zhao L.Zhu and X. Zhu. (2005). "From Virtualized Resources to VirtualComputing Grids: The In-VIGO System". In: *Complex Problem-Solving Environments for Grid Computing* 38.5, pp. 63–69. ISSN: 0018-9162. DOI: `http://doi.ieeecomputersociety.org/10.1109/MC.2005.175`.

Sakellariou, Rizos and Viktor Yarmolenko (2008). "Job Scheduling on the Grid: Towards SLA-Based Scheduling". In: *High Performance Computing and Grids in Action*. Ed. by Lucio Grandinetti. IOS Press. URL: `http://www.cs.man.ac.uk/~rizos/papers/hpc08.pdf`.

Schoonderwoerd, Ruud et al. (1996). "Ant-Based Load Balancing in Telecommunications Networks". In: *Adaptive Behavior* 2, pp. 169–207. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.4.3655`.

Server Support Staffing Ratios (2006). Tech. rep. URL: `http://www.computereconomics.com/custom.cfm?name=postPaymentGateway.cfm&id=1188&CFID=6797344&CFTOKEN=75835512`.

Shang, S. et al. (2010). "DABGPM: A Double Auction Bayesian Game-Based Pricing Model in Cloud Market". In: *Network and Parallel Computing*, pp. 155–164.

Sherwani, Jahanzeb et al. (2002). *Libra: An Economy driven Job Scheduling System for Clusters*. URL: `http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0207077`.

Siddiqui, Mumtaz, Alex Villazón, and Thomas Fahringer (2006). "Grid capacity planning with negotiation-based advance reservation for optimized QoS". In: *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. Tampa, Florida: ACM, p. 103. ISBN: 0-7695-2700-0. DOI: `http://doi.acm.org/10.1145/1188455.1188563`.

Smallen, Shava et al. (2000). *Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience*.

Smith, Warren, Ian Foster, and Valerie Taylor (2000). “Scheduling with Advanced Reservations”. In: *Parallel and Distributed Processing Symposium, International* 0, p. 127. ISSN: 1530-2075. DOI: `http://doi.ieeecomputersociety.org/10.1109/IPDPS.2000.845974`.

Somasundaram, T. et al. (2010). “CARE Resource Broker: A framework for scheduling and supporting virtual resource management”. In: *Future Generation Computer Systems* 26.3, pp. 337–347. ISSN: 0167-739X.

Sorensen, S.-A. and M. G.W. Jones (1992). “The clown network simulator”. In: Springer-Verlag. URL: `citeseer.ist.psu.edu/742869.html`.

Sørensen, Søren-Aksel and B Bauer (2003). “On the dynamics of the Kofels sturzstrom”. In: *Geomorphology*.

Sotomayor, Borja et al. (2009). “Capacity Leasing in Cloud Systems using the OpenNebula Engine”. In: Cloud Computing and Applications 2008 (CCA08).

Srinivasan, S. et al. (2002). “Characterization of backfilling strategies for parallel job scheduling”. In: *Parallel Processing Workshops, 2002. Proceedings. International Conference on*. IEEE, pp. 514–519. ISBN: 0769516807.

Stephen McGough, A., William Lee, and Shikta Das (2008). “A standards based approach to enabling legacy applications on the Grid”. In: *Future Gener. Comput. Syst.* 24 (7), pp. 731–743. ISSN: 0167-739X. DOI: `10.1016/j.future.2008.02.004`. URL: `http://portal.acm.org/citation.cfm?id=1377055.1377384`.

Stuer, Gunther, Vaidy Sunderam, and Jan Broeckhove (2005). “Towards OGSA compatibility in the H2O metacomputing framework”. In: *Future Gener. Comput. Syst.* 21.1, pp. 221–226. ISSN: 0167-739X. DOI: `http://dx.doi.org/10.1016/j.future.2004.09.011`.

Sugerman, Jeremy, Ganesh Venkitachalam, and Beng-Hong Lim (2001). “Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor”. In: *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, pp. 1–14. ISBN: 188044609X. DOI: `10.1145/265924.265930`. URL: `http://dx.doi.org/10.1145/265924.265930`.

Sullivan, Francis (2009). “Guest Editor’s Introduction: Cloud Computing for the Sciences”. In: *Computing in Science and Engineering* 11, pp. 10–11. ISSN: 1521-9615. DOI: `http://doi.ieeecomputersociety.org/10.1109/MCSE.2009.121`.

Sumby, Patrick (2006). *Temperature Mapping of the Lunar Surface Final Report*.

Sundararaj, A. and P. Dinda (2004). *Towards virtual networks for virtual machine grid computing*. URL: `citeseer.ist.psu.edu/article/sundararaj04towards.html`.

Sutherland, I. E. (1968). “A futures market in computer time”. In: *Commun. ACM* 11.6, pp. 449–451. ISSN: 0001-0782. DOI: `http://doi.acm.org/10.1145/363347.363396`.

Teng, F. and F. Magoulès (2010). “Resource Pricing and Equilibrium Allocation Policy in Cloud Computing”. In: *2010 10th IEEE International Conference on Computer and Information Technology (CIT 2010)*. IEEE, pp. 195–202.

Thain, Douglas, Todd Tannenbaum, and Miron Livny (2005). “Distributed computing in practice: the Condor experience.” In: *Concurrency - Practice and Experience* 17.2-4, pp. 323–356.

Thain, Douglas et al. (2001). “Gathering at the Well: Creating Communities for Grid I/O”. In: *SC Conference* 0, p. 21. DOI: `http://doi.ieeecomputersociety.org/10.1109/SC.2001.10023`.

The Grid Workloads Archive. URL: `http://gwa.ewi.tudelft.nl/pmwiki/`.

Tiwari, Ritesh Kumar, Vishal Dwivedi, and Kamalakar Karlapalem (2007). “SLA Driven Process Security through Monitored E-contracts”. In: *Services Computing, IEEE International Conference on* 0, pp. 28–35. DOI: `http://doi.ieeecomputersociety.org/10.1109/SCC.2007.109`.

Tsafrir, Dan, Yoav Etsion, and Dror G. Feitelson (2007). “Backfilling Using System-Generated Predictions Rather than User Runtime Estimates”. In: *IEEE Trans. Parallel Distrib. Syst.* 18.6, pp. 789–803.

Venugopal, Srikumar, Rajkumar Buyya, and Kotagiri Ramamohanarao (2006). “A taxonomy of Data Grids for distributed data sharing, management, and processing”. In: *ACM Comput. Surv.* 38 (1). ISSN: 0360-0300. DOI: `http://doi.acm.org/http://doi.acm.org/10.1145/1132952.1132955`. URL: `http://doi.acm.org/http://doi.acm.org/10.1145/1132952.1132955`.

Venugopal, Srikumar, Rajkumar Buyya, and Lyle Winton (2006). “A Grid service broker for scheduling e-Science applications on global data Grids: Research Articles”. In: *Concurr. Comput. : Pract. Exper.* 18 (6), pp. 685–699. ISSN: 1532-0626. DOI: `10.1002/cpe.v18:6`. URL: `http://portal.acm.org/citation.cfm?id=1133046.1133055`.

Vilajosana, Xavier, Ruby Krishnaswamy, and Joan Manuel Marquès (2009). “Design of a Configurable Auction Server for Resource Allocation in Grid”. In: *CISIS*, pp. 396–401.

Waldo, Jim (1999). “The Jini architecture for network-centric computing”. In: *Commun. ACM* 42.7, pp. 76–82. ISSN: 0001-0782. DOI: `http://doi.acm.org/10.1145/306549.306582`.

Waldspurger, C.A. et al. (1992). “Spawn: A Distributed Computational Economy”. In: *IEEE Transactions on Software Engineering* 18, pp. 103–117. ISSN: 0098-5589. DOI: `http://doi.ieeecomputersociety.org/10.1109/32.121753`.

Wallom, D.C.H. and A.E. Trefethen (2006). “Oxgrid, a campus grid for the university of oxford”. In: *Proceedings of the UK e-Science All Hands Meeting*.

Wei, G. et al. (2009). “A game-theoretic method of fair resource allocation for cloud computing services”. In: *The Journal of Supercomputing*, pp. 1–18. ISSN: 0920-8542.

Weissman, Jon B. and Xin Zhao (1998). “Scheduling parallel applications in distributed networks”. In: *Cluster Computing* 1.1, pp. 109–118. ISSN: 1386-7857. DOI: `http://dx.doi.org/10.1023/A:1019073113216`.

Wolinsky, David Isaac, Yonggang Liu, and Renato Figueiredo (2009). “Towards a uniform self-configuring virtual private network for workstations and clusters in grid computing”. In: *VTDC*

'09: Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing. Barcelona, Spain: ACM, pp. 19–26. ISBN: 978-1-60558-580-2. DOI: http://doi.acm.org/10.1145/1555336.1555340.

Wolski, R. et al. (2001a). "Analyzing market-based resource allocation strategies for the computational grid". In: *International Journal of High Performance Computing Applications* 15.3, p. 258. ISSN: 1094-3420.

Wolski, R. et al. (2001b). "G-commerce: Market Formulations Controlling Resource Allocation on the Computational Grid". In: *International Parallel and Distributed Processing Symposium (IPDPS)*. San Francisco: IEEEE. URL: http://www.cs.utk.edu/~plank/plank/papers/IPDPS-01.html.

Wolski, Rich et al. (2001c). "Analyzing Market-Based Resource Allocation Strategies for the Computational Grid". In: *Int. J. High Perform. Comput. Appl.* 15.3, pp. 258–281. ISSN: 1094-3420. DOI: http://dx.doi.org/10.1177/109434200101500305.

Wu, Fang, Li Zhang, and Bernardo A. Huberman (2005). "Truth-Telling Reservations." In: *WINE*. Vol. 3828. Lecture Notes in Computer Science. Springer, pp. 80–91. ISBN: 3-540-30900-4. URL: http://dblp.uni-trier.de/db/conf/wine/wine2005.html#WuZH05.

Xavier, Percival, Bu-Sung Lee, and Wentong Cai (2003). "A Decentralized Hierarchical Scheduler for a Grid-Based Clearinghouse". In: *Parallel and Distributed Processing Symposium, International* 0, 269b. ISSN: 1530-2075. DOI: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2003.1213487.

Xhafa, Fatos and Ajith Abraham (2010). "Computational models and heuristic methods for Grid scheduling problems". In: *Future Generation Computer Systems* 26.4, pp. 608–621. ISSN: 0167739X. DOI: 10.1016/j.future.2009.11.005. URL: http://dx.doi.org/10.1016/j.future.2009.11.005.

Xiao, Peng and Zhi gang Hu (2009). "Relaxed resource advance reservation policy in grid computing". In: *The Journal of China Universities of Posts and Telecommunications* 16.2, pp. 108 –113. ISSN: 1005-8885. DOI: DOI:10.1016/S1005-8885(08)60213-7. URL: http://www.sciencedirect.com/science/article/B8H14-4W6F34M-S/2/8526c7dcd3a75a06c53ec93b3eb133cd.

Xiong, Kaiqi and Harry Perros (2006). "Trust-based Resource Allocation in Web Services". In: *Web Services, IEEE International Conference on* 0, pp. 663–672. DOI: http://doi.ieeecomputersociety.org/10.1109/ICWS.2006.135.

Xu, Ming Q. (2001). "Effective Metacomputing using LSF MultiCluster". In: *ccgrid* 00, p. 100. DOI: http://doi.ieeecomputersociety.org/10.1109/CCGRID.2001.923181.

Yang, C.T. et al. (2005). "On Construction of a Large File System Using PVFS for Grid". In: *Parallel and Distributed Computing: Applications and Technologies*, pp. 231–249.

Yeo, Chee Shin and Rajkumar Buyya (2005). "Service Level Agreement based Allocation of Cluster Resources: Handling Penalty to Enhance Utility". In: *Proceedings of the 7th IEEE International*

Conference on Cluster Computing (Cluster 2005). Boston, MA, USA: IEEE Computer Society: Los Alamitos, CA, USA. ISBN: 0-7803-9486-0. DOI: `10.1109/CLUSTR.2005.347075`.

— (2006). "A taxonomy of market-based resource management systems for utility-driven cluster computing". In: *Softw. Pract. Exper.* 36.13, pp. 1381–1419. ISSN: 0038-0644. DOI: `http://dx.doi.org/10.1002/spe.v36:13`.

— (2007). "Pricing for Utility-driven Resource Management and Allocation in Clusters". In: *International Journal of High Performance Computing Applications* 21.4, pp. 405–418. ISSN: 1094-3420. DOI: `10.1177/1094342007083776`.

Yu, Jia and Rajkumar Buyya (2005). "A taxonomy of scientific workflow systems for grid computing". In: *SIGMOD Rec.* 34.3, pp. 44–49. ISSN: 0163-5808. DOI: `http://doi.acm.org/10.1145/1084805.1084814`.

Zasada, S.J. and P.V. Coveney (2009). "Virtualizing access to scientific applications with the Application Hosting Environment". In: *Computer Physics Communications* 180.12. 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures, pp. 2513 –2525. ISSN: 0010-4655. DOI: `DOI:10.1016/j.cpc.2009.06.008`. URL: `http://www.sciencedirect.com/science/article/B6TJ5-4WHFD89-1/2/46719de5e084a27384fb242d08233c05`.

Zhang, Li and Danilo Ardagna (2004). "SLA based profit optimization in autonomic computing systems". In: *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*. New York, NY, USA: ACM, pp. 173–182. ISBN: 1-58113-871-7. DOI: `http://doi.acm.org/10.1145/1035167.1035193`.

Zhang, Xuechai, J.L. Freschl, and J.M. Schopf (2003). "A performance study of monitoring and information services for distributed systems". In: pp. 270 –281. DOI: `10.1109/HPDC.2003.1210036`.

Zhang, Xuehai and J.M. Schopf (2004). "Performance analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2". In: pp. 843 –849. DOI: `10.1109/PCCC.2004.1395199`.

Zhang, Yanyong et al. (2000). "A simulation-based study of scheduling mechanisms for a dynamic cluster environment". In: *ICS '00: Proceedings of the 14th international conference on Supercomputing*. Santa Fe, New Mexico, United States: ACM, pp. 100–109. ISBN: 1-58113-270-0. DOI: `http://doi.acm.org/10.1145/335231.335241`.

Zhang, Yanyong et al. (2001). "An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling, and Migration". In: *JSSPP '01: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*. London, UK: Springer-Verlag, pp. 133–158. ISBN: 3-540-42817-8.

Zhao, Ming, Jian Zhang, and Renato J. Figueiredo (2006). "Distributed File System Virtualization Techniques Supporting On-Demand Virtual Machine Environments for Grid Computing". In: *Cluster Computing* 9.1, pp. 45–56. ISSN: 1386-7857. DOI: `http://dx.doi.org/10.1007/s10586-006-4896-x`.