THE UNIVERSITY OF CHICAGO

TAKING ADVANTAGE OF USAGE SERVICE LEVEL AGREEMENTS FOR BETTER GRID RESOURCE SCHEDULING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
CATALIN LUCIAN DUMITRESCU

CHICAGO, ILLINOIS
JUNE 2007

To my family and friends

Abstract

Challenging issues concerning resource usage can arise within virtual organizations (VOs) that integrate participants and resources spanning multiple physical institutions. Participants may wish to delegate to one or more VOs the right to use certain resources subject to local policy and service level agreements; each VO then wishes to use those resources subject to VO policy. In this dissertation I propose, design, build, and evaluate a different approach for controlled resources sharing in large distributed systems based on usage service level agreements (uSLAs). The proposed model is targeted for large and dynamic distributed environments and is itself distributed in order to cope with large communities of users that reside in different administrative domains. Agreements and Service level Agreements are not new concepts at this time. However, applying these concepts in a large scale Grid that makes scheduling and resource sharing effective is difficult. Without such an agreement based resource sharing mechanism in place, existing Grid scheduling, centralized or distributed, either do not scale well, or are not effective due to a high overhead of gathering up-to-date resource availability information. My thesis is that the explicit representation, enforcement, and management of uSLAs can serve as an objective organizing principle for such systems. uSLAs express how resources must be used over time intervals and represent a novelty for the Grids. The concept comes from the networking domain, where bandwidth is allocated based on specific rules.

The main objective is resources neither to remain idle when there are available workloads for execution, nor a VO to consume more computing resources than provided. In support of this thesis, my contributions are as follows. First, I propose new mechanisms for uSLA specification and enforcement at various levels and perform experimental measurements of the improvements that these mechanisms can enable in different environments and for different workloads. Second, based on the real deployments, I introduce a method for determining uSLAs via observation rather than specification, that is, an algorithm that a client can use to determine automatically the uSLA that is delivered by a resource in practice. Third, I introduce GangSim, a simulator for Grid scheduling studies that allows uSLAs to be specified and simulated at different levels, as well as automated performance measurements. And fourth, I present GRUBER, a Grid resource scheduling prototype and architecture that allows uSLAs to be specified by site, VO, and, group administrators.

The results show that uSLAs can be implemented with success and I provide insights into the performance and utility of the uSLA mechanisms. For example, I show that for real workloads on a real Grid, the measured response time is 2.67 times higher and site utilization is up to ten times higher than a simple round robin strategy. In the same case, the response time is 1.16 times higher and site utilization is equal compared to an “optimistic” approach that sends jobs to recently responsive sites.

Committee Members

Ian Foster, Director

Mathematics and Computer Science

Argonne National Laboratory and The University of Chicago

Anne Rogers, Professor

Computer Science Department

The University of Chicago

Asit Dan, Ph.D.

IBM Software Group

Somers, NY

Acknowledgments

Firstly, I would like to thank my committee members. Ian Foster, my PhD advisor, provided the best support for my endeavor during all this time. Also, every time I doubted myself Ian gave me good reasons to continue. During the past two years, Anne provided focus and guidance and gently pushed me to extend the work presented in this dissertation. During the internship at IBM and in the years afterwards, Asit had an important role in shaping the work described in this dissertation. Moreover, Ian, Anne, and Asit have had a major impact on this piece of research and on shaping my life during the past years.

Secondly, I would like to thank Robert Gardner, Marco Mambelli, Michael Wilde, Jens-S. Vöckler, and Ioan Raicu for their insights and discussions in support of this work. I also thank the Grid3 project. This work was supported in part by the NSF Information Technology Research GriPhyN project, under contract ITR-0086044.

Thirdly, I would like to thank Dick Epema, Alexandru Iosup, Lex Woulters, and Hui Li for the help on better understanding the importance and size of the traces analyzed in this dissertation. Some of the results and observations reported here were possible only with the support from the CoreGRID IST project $n^0$004265, funded by the European Commission.

Some of the graphs in this thesis were created using the RRDtool [1, 2].

Table of Contents

List of Figures

List of Tables

CHAPTER ONE

INTRODUCTION

In this dissertation I propose, design, build, and evaluate an approach for controlled resources sharing in large distributed systems. The proposed model is targeted for large and dynamic distributed environments and is itself distributed in order to cope with large communities of users that reside in different administrative domains. Grids [3, 4] and Peer-to-Peer systems [5-11], as real-world examples, provide the requirements for this work and serve as testbeds to evaluate potential solutions in realistic settings. I focus on Grid computing because it enables participants to share many types of resources: CPUs, disk, network or other complex services.

The thread shared by most Grid systems is *cooperative computing* [12]. The goal of these systems is to provide large-scale, flexible, and secure resource sharing among dynamic collections of individuals, institutions, and resources, also referred as virtual organizations (VOs) [4]. In such settings, users from multiple administrative domains pool available resources to harness their aggregate power and to benefit from the increased computing power and the diversity of these resources, especially when their applications are customized for a specific computing configuration (i.e., 64-bit architectures vs. 32-bit architectures, ring vs. star network topology).

Resource sharing within large distributed systems that integrate participants and resources spanning multiple physical institutions raises challenging issues [13]. Physical institutions may wish to delegate to one or more participants the right to use certain resources subject to local preferences and various agreements; each participant then wishes to enable those resources subject to their own policy. Mechanisms for supporting controlled resource sharing must be designed to allow cooperative systems to provision resources based on pre-negotiated agreements and on providers' preferences.

The increased scale of distributed systems calls for minimizing the needs for human supervision and for automating as many management tasks as possible. For example, in a system with over 10,000 nodes, new settings may occur thousands of times more often than when no resources are shared. At the same time, the complexity of necessary services will increase with the scale of the system. For example large and distributed systems require resource discovery and brokering services.

Increasing scale in cooperative computing also makes performance and reliability challenging. Centralized systems are unlikely to rise to this challenge of serving as a single unified management decision point for hundreds to thousands of jobs and sites, and they can become a bottleneck in terms of both reliability and performance. Additionally, in a wide area network, where short and transient failures often occur, a single decision point can become inaccessible for varying time periods. Distributed services can alleviate these challenges and improve availability.

Current solutions for controlling resource access in large scale distributed systems focus mainly on access control [14, 15], but other groups have started pursuing various paths for controlled resource sharing [16-22]. Finer access control mechanisms focus on allowing resource providers to express additional conditions about access and delegate partial control to other entities. For example, a Community Authorization Service (CAS) for access control policy management allows resource providers to maintain ultimate authority over their resources, but spares them from day-to-day policy administration tasks (e.g. adding and deleting users, modifying user privileges) [14]. Access control dictates the operations an entity is entitled to perform on certain resource without any further restrictions once access is granted. Other methods focus on economic models or matchmaking for controlled resource provisioning. In such cases, mini-markets are built for resource brokering and provisioning [16]. Another approach, *Service level agreements* (SLAs) [22, 23], focuses on establishing consumer-provider relationships concerning how resources must be consumed. Such relationships can be designed by bi-lateral rules that are driven by the internal policies that govern an institution.

The difference of my work from access control mechanisms consists in its support for finer control about what fraction of resources a user can use after access was granted. Currency-based mechanisms also allow for such finer access control, but they offer instead a homogeneous access mechanism without enforcing any other rules pertaining to user characteristics (like physical location or a supporting VO). Agreements and Service level Agreements are not new concepts at this time [21, 24,

25]. However, applying these concepts in a large scale Grid environment (multi-domain resource sharing) that makes scheduling and resource sharing effective is difficult. Without such an agreement based resource sharing mechanism in place, existing Grid scheduling, centralized or distributed [18, 19, 37, 59, 60, 137], either do not scale well, or are not effective due to a high overhead of gathering up-to-date resource availability information. Agreement-based sharing focuses on establishing bi-lateral consumer-provide relations. When many players are involved, the establishment of bilateral agreements among all parties is difficult. Thus, after a provider has established all the agreements he wants, he could use uSLA mechanisms to express them and ensure their enforcement at the Grid level.

In the networking domain, usage service level agreements (uSLAs) are used to address the problem of bandwidth allocation based on specific rules. Such policies are specified by network administrators and contain the rules for handling different types of traffic. In this domain, a simple usage policy example is “Email traffic is only allowed from outside the company’s servers only from a special mail gateway.” [26-29] This technique cannot be applied directly to Grids without addressing the problem of multi-type resources (CPU, disk, services). In this dissertation I address how a refined and extended concept of usage policy can be applied with success in a Grid setting.

In this chapter I provide an initial scenario for the resource sharing problem, a description of the research problem and a succinct introduction to the supporting frameworks and results achieved during this work. I finish this chapter with a discussion of my contributions and a roadmap of how I explore the explicit

representation, enforcement, and management of uSLAs in large distributed environments.

1.1 Envisaged Scenarios

The environments I target comprise potentially large numbers of resources, resource providers, and virtual organizations (VOs) [30, 31]. For example, in the sciences, hundreds of institutions and thousands of individual investigators may collectively control tens or hundreds of thousands of computers and associated storage systems. Each individual investigator and institution may participate in, and contribute resources to, multiple collaborative projects that can vary widely in scale, lifetime, and formality. At one end of the spectrum, two collaborating scientists may want to pool resources for the purposes of a single analysis. At the other extreme, the major physics collaborations associated with the Large Hadron Collider [32] encompass thousands of physicists at hundreds of institutions, and need to manage workloads comprising dynamic mixes of work of varying priority, some requiring the efficient aggregation of large quantities of computing and storage.

Figure 1.1 shows a high-level model of resource allocation. In the two VOs (squares) and three sites (circles), shaded elements indicate the compute (C) and storage (S) resources allocated to each VO at each site. Sites and VOs share resources by defining *how* resource usage takes place in terms of *who, what, where,* and *when* it is allowed. VOs and SLAs can, of course, be nested [33].

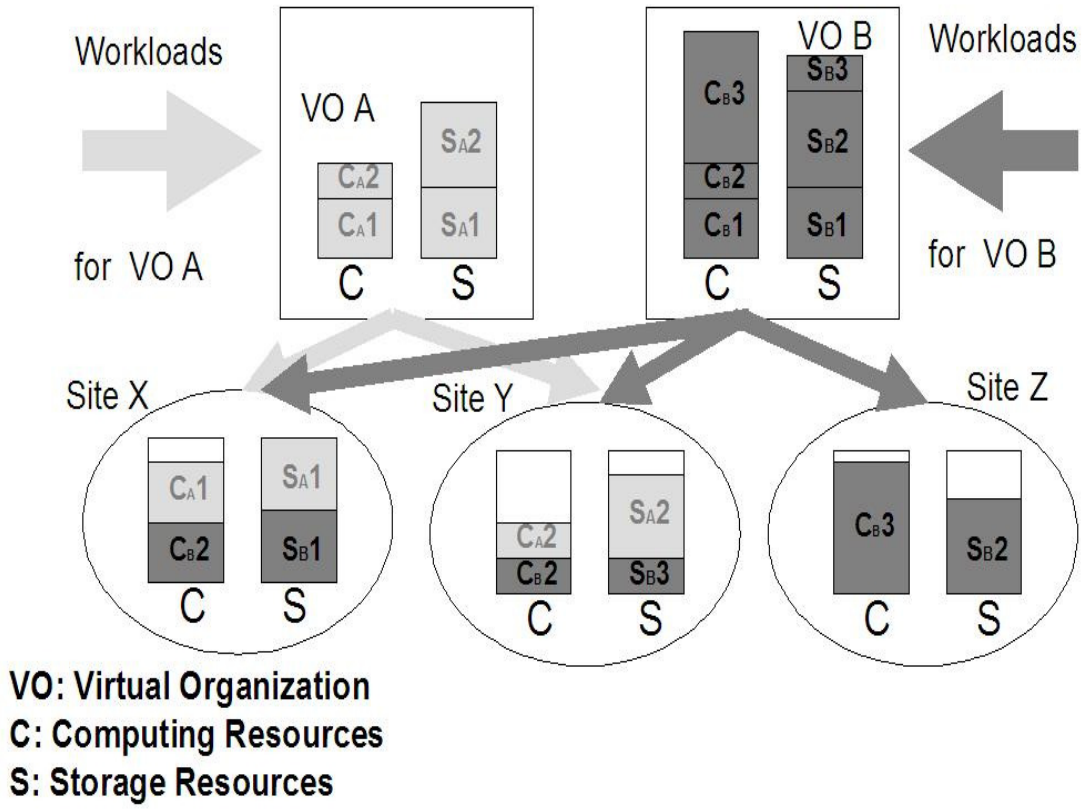


Figure 1.1: Resource Allocation Schematic. VO A's computing power is aggregated from Sites X and Y, while VO B's computing power is aggregated from Sites X, Y and Z. Similarly for disk space.

Users submit workloads that may comprise many thousands of independent or loosely coupled tasks [13, 34-37]. I use an introductory example (see Figure 1.2) to show how an uSLA can be used to guide resource allocations in such scenarios, and to illustrate some of the issues that must be addressed when implementing uSLAs. Assume that provider P has R computing resources available and wants to provide a portion of these resources (R_1) to consumer C_1 for a period of one month. The provider must be able to express and enforce this agreement. We must also be concerned with exactly how this agreement is to be interpreted. The resources in question might be dedicated to C_1 or, alternatively, P might make them available to others when C_1 is not using them. In addition, if C_1 is allowed to acquire more than its allocation when resources are not being used for other purposes, then this "over allocation" may or may not result in C_1's allocation being reduced later in the month [33]. C_1, in turn, wants to discover the allocation made by P, to interpret this agreement in terms of its semantics, and to use the resulting information when making scheduling decisions: it may, for example, choose to send tasks to another provider rather than P, if its allocation at P is exhausted. C_1 may also want to monitor the resources that it obtains from P, to verify that P is adhering to the uSLA that has been negotiated. (In other contexts, monitoring can also be used to infer the uSLA that is place, when this information is not made available, or proves to be incorrect).

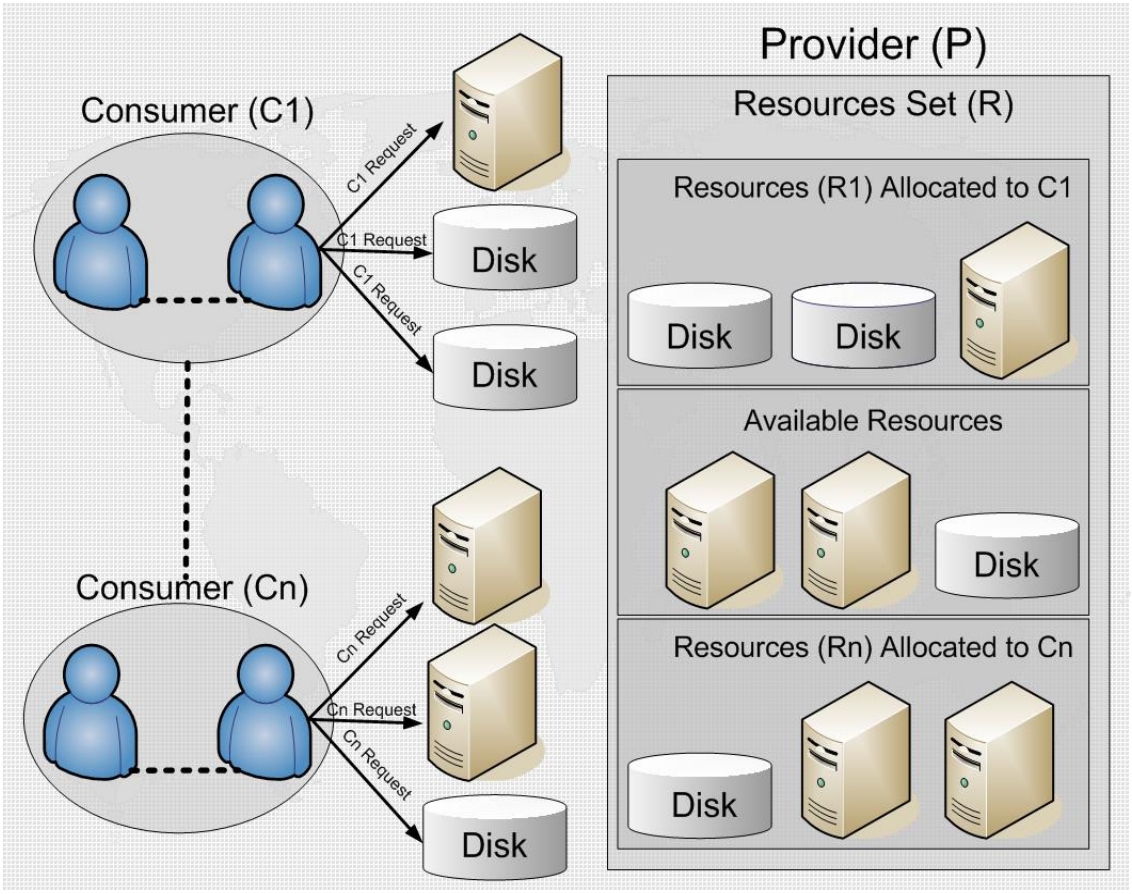


Figure 1.2: uSLA Introductory Example

Thus, an uSLA [38, 39] expresses a relationship between a resource provider (a site or VO) and a resource consumer (a VO or user). Brokers, or directly, consumers aggregate resources acquired from different providers, and orchestrate distributed computations to use those aggregated resources efficiently. In this context, a broker represents an entity that aggregates resources from various providers and advertises them to other consumers.

Starting from the above example, I define an uSLA as representing a provider statement about a contract that governs how its resources are to be allocated to a specific resource consumer. In a typical scenario, individual resource providers negotiate uSLAs with relevant consumers or brokers to establish what resources are available for use. Policy makers who participate in such collaborations define usage policies involving various levels of resource sharing. A VO in its role as both resource consumer (from sites or other VOs) and provider (to its consumers: users or groups within the VO) acts as a broker for a set of resources.

1.2 Refining the Research Problem

Running workloads in a Grid environment without any knowledge about the employed resource sharing policies can be a challenging problem. For example, Grid3 [13] represents a multi-virtual organization environment, is composed of 30 sites around US and sustains production level services required by various physics experiments [40]. Usually, these participating sites are the OSG/Grid3 resource providers and their resources are made available under various conditions not captured

by any monitoring tool (i.e., VO priorities, local usage policies, etc). In this environment, users often face the problem of high latencies for their jobs even though various monitoring tools show the selected sites as partially available. Thus, the question "*How should usage policies for scheduling be represented and used?*" becomes important to address. Important challenges range from the lack of automated mechanisms for uSLA discovery, publication, or interpretation [13] to the complexity of the uSLA operations to be performed to satisfy the requirements [10, 41] of many resources and users. Additionally, controlled resource sharing mechanisms may introduce too much overhead to justify their deployment in real environments.

I make two assumptions in this dissertation. First, providers have local policies about how their resources are used, while also focusing on achieving higher gains for these resources (i.e., utilization). Second, consumers want to achieve better performance for their applications and to acquire as many resources as possible when their applications require a lot of computational power (for example, the physicists participating in the LCG project [32]).

In a large distributed environment, both computing resources and site management solutions are heterogeneous. Thus, determining the usage policies at each individual site might not be trivial. Also, these policies are always specified in terms of specific local resource manager syntax and semantics. In cases where such policies are not even specified, local sites would require new usage policy mechanisms for enforcement when joining a large computing collaboration. Thus we must answer the question *"How*

can a system determine and enforce providers' policies, if these policies not specified explicitly?"

The last question, *"What benefit can be derived from relying on uSLAs for controlled resource sharing?"* captures the notion that the participants in a cooperative environment are interested in maximizing their own benefits. Thus, the sharing mechanisms must provide real gains for all participants to be successful.

1.3 Supporting Frameworks and Experiments

This dissertation describes a novel approach to controlling resource sharing in large distributed systems. To demonstrate the practicality of this model, I have designed a notation for uSLA specification and defined four uSLA semantics; simulated the four uSLA semantics using a resource scheduling simulator named GangSim [42]); evaluated a decentralized solution for uSLA management based on the GRUBER [43] framework for large distributed and dynamic environments; and demonstrated in practice how uSLA-based resource sharing works using the GRUBER framework.

In the rest of this section I introduce the two supporting tools developed specifically to evaluate uSLA-based model and present an overview of the results that support this dissertation.

1.3.1. Frameworks

The first tool, GangSim [42], is a VO-centric Grid simulator I developed for the scheduling studies presented in this dissertation. GangSim simulates a context where

hundreds of institutions and thousands of individuals collectively control hundreds to thousands of computers and associated storage systems.

GangSim is a discrete simulator that periodically evaluates the state of all simulated components: sites, VOs, schedulers (internal, external and data), monitoring data points, site policy enforcement points and VO policy enforcement points. GangSim also evaluates costs associated with the simulated components: time to enter the scheduling queues, time for site assignment, time for site transfer (network allocation and transfer for the executable), time for node assignment, and time for job transfer (network allocation and transfer for the executable and data) [44].

The main purpose of GangSim is to provide support for simulating the combination of uSLAs, scheduling policies and various workloads on certain Grid architectures and to measure various metrics, such as: resource utilization, response times, local and VO policy violations, and queue times when failures and costs are simulated.

The second tool, GRUBER [43], is a framework for uSLA discovery, resource management and job routing that I developed for deployment in Grid environments. It discovers and uses sites' uSLAs for better resource scheduling. GRUBER focuses on computing resources such as computers, storage, and networks; providers may be either individual scientists or sites; and VOs are collaborative groups, such as scientific collaborations. GRUBER adds to the classical Grid concrete support for discovering, specification, and enforcement of uSLAs. Also, GRUBER supports uSLA management at several levels in a Grid, from the site level up to the end user level [43]. Whenever a management decision must be performed, the uSLAs from several levels are composed

to compute the actual resource allocation for the requesting user. To cope with large and dynamic Grids, several decision points can be deployed in a distributed manner and they cooperate for accurate decision making [45]. When a Grid scheduler is missing, GRUBER can also play the role of a basic Grid scheduling infrastructure.

1.3.2. Results

The results presented in this dissertation can be grouped into three main categories: analyses by means of simulations of uSLA semantics and infrastructure models, analyses of uSLA-based resource sharing and models with support for decentralization and dynamic Grids, and analyses of previous concepts for job submission over OSG/Grid3 [13].

I performed extensive experiments using GangSim to evaluate alternative uSLAs, architecture models, and mechanisms for collecting, publishing, expressing, and enforcing policies at various levels in a Grid. I also devised and analyzed the behavior of four uSLAs semantics — *no-limit*, *fixed-limit*, *extensible-limit*, and *commitment-limit* — in conjunction with several scheduling policies. The simulation results showed that the *commitment-limit* semantics performs best in terms of both resource utilization and response time.

My results obtained with GRUBER demonstrate that taking into account usage policies where they exist and performing uSLAs-based scheduling allow both resource consumers and providers to achieve higher performance (smaller response times and better resource utilization, respectively). I used this framework to simulate a Grid ten times larger than OSG/Grid3 on PlanetLab [46] under a constant workload of at least

one job per second from 120 submission hosts. These experiments show that three to five decision points are sufficient to handle the decisions required for such a Grid [45].

Finally, I use the BLAST workloads [43] over OSG/Grid3 [13] to show that uSLAs can be used successfully in practice and to show the expected performance for such situations [40], as reported in Section 6.4.3 of this dissertation. In the OSG/Grid3 case, the measured response time is 2.67 times higher and site utilization is up to ten times higher than a simple round robin strategy. In the same case, the measured response time is 1.16 times higher and site utilization equal compared to an "optimistic" approach that simply sends jobs to recently responsive sites.

1.4 Contributions

This dissertation introduces an uSLA-based resource sharing model for large distributed environments. The contributions of this work are:

1. New mechanisms for uSLA specification and enforcement at various levels and experimental measurements that demonstrate the performance improvements that these mechanisms can enable in different environments and for different workloads.
2. A method for determining uSLAs via observation rather than specification, that is, an algorithm that a client can use to determine automatically and dynamically the uSLA that is delivered by a resource in practice.
3. A Grid resource scheduling architecture that allows uSLAs to be specified by site, VO, and, group administrators. Also, a software framework, called GRUBER, that

implements that architecture and permits experimentation with alternative implementations of different functions.

4. A simulator, GangSim, for Grid scheduling studies that allows uSLAs to be specified and simulated at different levels (sites, VOs, and groups), as well as automated performance measurements.

The contributions for each chapter are outlined next. First, I discuss related work in Chapter 2. Second, I introduce the uSLAs problem and discuss representations for uSLAs in the Grid environment (Chapter 3). Third, I present detailed descriptions of the GangSim simulator and the GRUBER framework (Chapter 4 and 5). Fourth, I present experimental results gathered for two main scenarios, site level uSLA-based resource management and those gathered using GRUBER on the OSG/Grid3 testbed (Chapter 6). The dissertation ends with conclusions and future research directions (Chapter7).

1.5 Metrics

This section defines metrics used in this dissertation to evaluate and compare different mechanisms for controlled resource sharing.

Aggregated resource utilization (**Util**) is defined as the ratio of the CPU time actually consumed by the N jobs executed during the period considered ($\mathbf{\Sigma^{N}_{i=1}\, ET_i}$) to the total CPU time available over that time:

$$\mathbf{Util = \Sigma^{N}_{i=1}\, ET_i \,/\, (\#\_of\_CPUs * \Delta t)}$$

Util is designed to capture both the enforcement of consumers' preferences and how well a provider's resources were used.

Total job completion per site, VO or overall (**Comp**), measures the total number of jobs from a given set that are completed in a certain interval:

$$\textbf{Comp = Completed\_Jobs / \#\_of\_Jobs * 100.00}$$

This metric is designed to quantify the imbalance introduced by a given scheduling policy. It is useful for assessing the overall performance of the scheduling strategy.

Average site response time (**Delay**) represents the average time, per job ($\mathbf{DT_i}$), that elapses from when the job arrives at a resource provider's queue until it starts (the time between when a request is made and when the job starts at the site level):

$$\textbf{Delay} = \Sigma^{N}_{i=1}\ \mathbf{DT_i} / \textbf{\#\_of\_Jobs}$$

This metric is designed to order sites in a Grid function of their responsiveness in scheduling jobs.

Average Grid response time (**Response**) is computed as the average time per job that elapses from the job submission to an external scheduler queue until it starts ($\mathbf{RT_i}$), in other words, the time between when a request is made at the Grid level and when the job starts on a site:

$$\textbf{Response} = \Sigma^{N}_{i=1}\ \mathbf{RT_i} / \mathbf{N}$$

Response combined with the number of completed jobs is designed to allow users to quantify their satisfaction.

Average starvation factor (**Starv**) represents the ratio of the resources requested and available, but not provided to a user, to the resources consumed by the user ($\mathbf{ET_i}$), where *i* represents a site number (N). I compute this quantity as follows, where $\mathbf{ST_i}$ is

the CPU resource requested by a user but not provided, and $\mathbf{RT_i}$ is the total resources available:

$$\mathbf{Starv = \Sigma^{N}_{i=1} (MIN (ST_i, RT_i)) / \Sigma^{N}_{i=1} ET_i}$$

This metric is designed to measure how well a scheduling approach provides resources when available.

uSLA violation ratio (**Violation**) represents the ratio of the CPUs consumed by users ($\mathbf{BET_i}$) over the allocation interval to total the CPU power:

$$\mathbf{Violation = \Sigma^{N}_{i=1} BET_i / (\#\_of\_CPUs * \Delta t)}$$

This metric is designed to quantify a users' satisfaction under a specific uSLA over a given time interval $\boldsymbol{\Delta t}$.

I define the *scheduling accuracy* (**Accuracy**) for a specific job ($\mathbf{SA_i}$) as the ratio of free resources at the selected site to the total free provided resources over the entire Grid. **Accuracy** is then the aggregated value of all scheduling accuracies measured for each individual job:

$$\mathbf{Accuracy = \Sigma^{N}_{i=1} SA_i / N}$$

Accuracy is designed to measure how well a distributed brokering system deals with partial information.

Error per job (**EPJ**) is the number of failures per job. This metric is used to quantify the improvement in job execution over OSG/Grid3 between the two sets of experiments presented in Chapter 6.

Throughput is defined as the number of requests completed successfully by the service per unit time.

Replan is the number of re-scheduling operations performed during a considered workload execution or time interval.

Time is the total execution time for a workload considered for any of the experiments in this dissertation.

Speedup is the serial execution time to the grid execution time for a workload, and, finally, **Spdup75** is the serial execution time to the Grid execution time for 75% of a workload.

CHAPTER TWO

RELATED WORK

In this chapter I introduce in more detail the Grid computing domain, as defined by Foster et al. [3, 4, 47], and, next, I describe the main approaches and available systems for controlled resource sharing in such environments. The chapter also includes a survey of existing Grid simulators and their limitations that supported the necessity for building GangSim, a means for uSLA-based scheduling studies. The chapter ends with taxonomy of the policies used for controlled resource sharing in this domain.

2.1. Introduction

The term *Grid* was coined in the mid 1990s to denote a distributed computing infrastructure for advanced science and engineering. Grid computing defined through a series of excerpts introduced by Foster et al. [4]:

1. "Grid computing has emerged as an important new field, distinguished from conventional distributed computing by its focus on large-scale resource sharing, innovative applications, and, in some cases, high-performance orientation."

2. "... the 'Grid problem, which I define as flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources — what I refer to as virtual organizations. In such settings, I encounter unique authentication, authorization, resource access, resource discovery, and other challenges. It is this class of problem that is addressed by Grid technologies."
3. "The real and specific problem that underlies the Grid concept is coordinated resource sharing and problem solving in dynamic, multi-institutional VOs. The sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resource brokering strategies emerging in industry, science, and engineering. This sharing is, necessarily, highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs. A set of individuals and/or institutions defined by such sharing rules form what we call a virtual organization (VO)."

In a nutshell, Grid computing is about large scale resource sharing, innovative applications, and high performance computing [4]. It is meant to offer flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources, namely VOs. Figure 2.1 below gives a glimpse into the complex nature of Grid environments; it depicts VOs (blue clouds) interconnected at both the physical

layer (via networks) and at the abstract layer (via service layer agreements). Each VO has resources (i.e. direct access to computers, software, and data) and users. Based on the agreements among the various VOs, users can locate and share resources that are part of different VOs and could be physically located anywhere in the world.

The Grid provides a means for resource sharing distributed in different administrative domains. The main topics of research are:

Resource Management: Developing uniform and scalable mechanisms for naming, locating, and allocating computational and communication resources in distributed systems is a difficult problem. Usually, resources are hidden by various layers of software management and offering a transparent interface for utilization becomes a difficult problem to address in practice.

Data Management and Access: An important element is the design and production of infrastructure-level architecture for data management, which are called the data grid. Many applications that are intended for grids are data intensive, in the sense that they require large amount of input and output data for normal execution. Data placement is a difficult problem; because it has to take in account also the network link capacities and local CPU power. Also, data scheduling must proceed (or complement) job scheduling to avoid job starvations due to missing data.

Information Services: Requirements, designs, and prototypes of Grid information service are critical in large environments. Such services provide useful information about what resources are available or how resources must be used and allocated. Also,

these services become an enabler for dynamic application configuration and adaptation in dynamic environments where resources join and leave the environment at a fast rate.

Security: Secure group communications, management of trust relationships, and developing new mechanisms for fine-grained access control are important for grid applications. To ensure authenticity without the physical necessity of holographic signatures, signing cryptographic mechanisms are developed and provided for messages and documents. Encrypting the entire document with a receiver's public key assures that only the receiver might be able to read the document's content. The purpose of such signing / encrypting, then, is to guarantee that any attempt to modify / read the content of a document becomes practically impossible. Thus, key possession and protection become an important issue in a world that lives in digital environments and needs confidentiality or/and privacy.

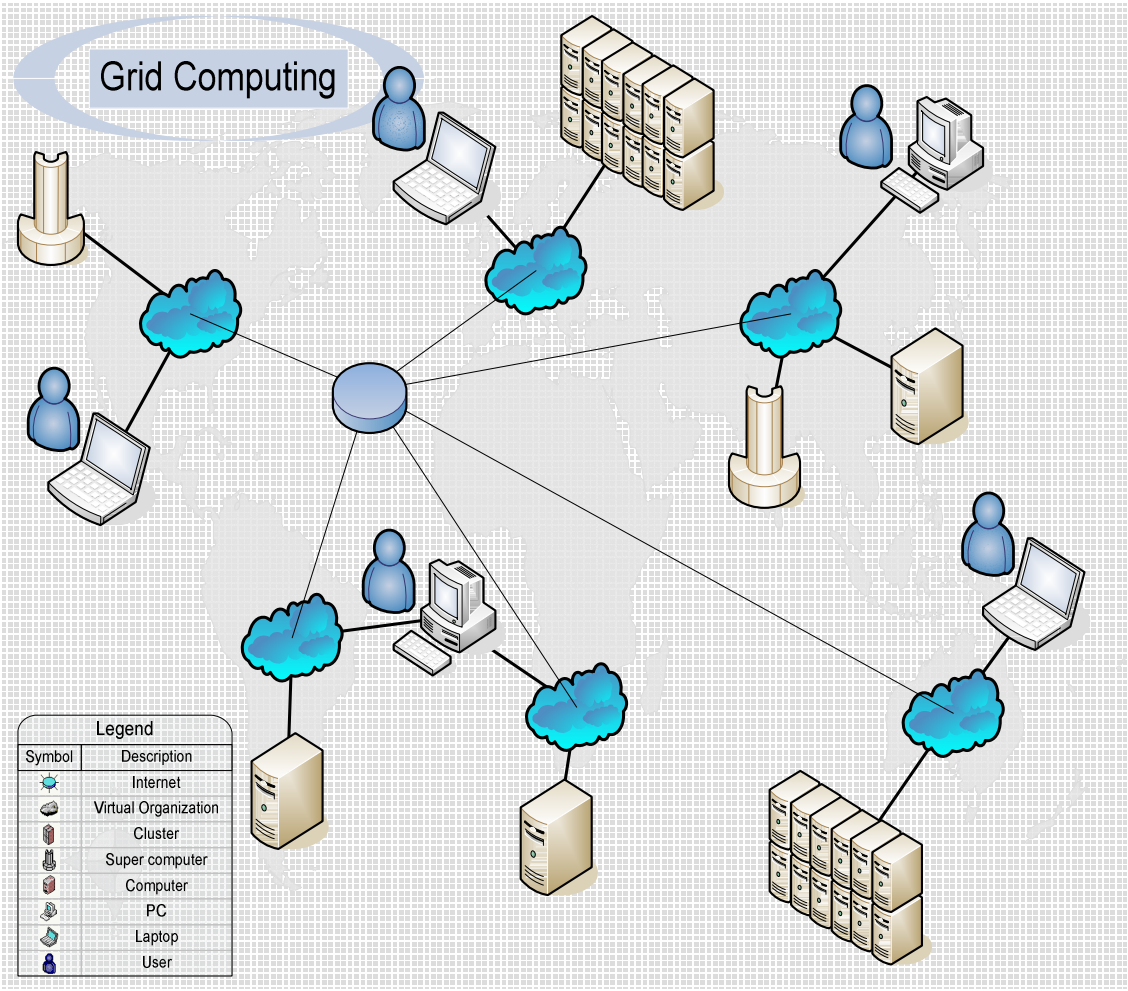


Figure 2.1: Grid Computing Overview

Motivated by a desire to support better resource management and access services, more and more effort is invested in research and development of systems for delivering management services over national and global scopes [48]. Current solutions for controlling resource utilization in large scale distributed systems focus mainly on access control [14, 15], but other groups have started pursuing various paths for controlled resource sharing [16-22]. Next, I detail some of the most common Grid technologies for resource management and access.

2.2. Resource Management Solutions for Grids: A Survey

I describe in this section four well known efforts for controlled resource management in Grids, plus three additional infrastructures targeted at resource management. Each of these solutions is pursued by different communities or groups and have large acceptance for the Grid community [31, 49].

2.2.1 Community Authorization Service (CAS)

Community Authorization Service (CAS [14, 15]) represents one of the efforts pursued by the Globus team for providing controlled access to Grid resources. CAS builds on X.509 and Certificate Authority (CA) concepts in order to support controlled resource sharing for resource providers that specify course-grained access control policies, and delegating fine-grained access control policy management to the community itself.

X.509 [50, 51] defines a centralized control framework for providing authentication services by using cryptographic techniques. There are three main roles required by this

approach: Certificate Authorities, Subscribers and Users. An additional entity is represented by a Naming Authority (NA). CA [52] is the entity recognized by several principals that controls authentication mechanisms and the management of certificates. Principals trust a CA by knowing (correct: believe they know) its correct public key and accepting unconditionally as authentic all certificates signed with this key. Because the verification process requires as information only the public key of the signing CA and certificates, the entire process can be completed without the permanent presence of the CA online. The only issue is the acquisition of the correct public key of the trusted CA.

By means of CAS, resource providers maintain ultimate authority over their resources but are spared from day-to-day policy administration tasks (e.g. adding and deleting users, modifying user privileges). CAS is designed as a centralized server that is initiated for a community: a community representative acquires a GSI credential to represent that community as a whole, and then runs a CAS server using that community identity. Later, resource providers grant privileges to the community. Each resource provider verifies that the holder of the community credential represents that community and that the community's policies are compatible with the resource provider's own policies. Once a trust relationship has been established, the resource provider then grants rights to the community identity, using normal local mechanisms. Also, a user uses the credentials from the CAS to connect to the resource with any normal Globus tool (e.g., GridFTP [53]). The resource then applies its local policy to determine the amount of access granted to the community, and further restricts that access based on

the policy in the CAS credentials. This serves to limit the user's privileges to the intersection of those granted by the CAS to the user and those granted by the resource provider to the community. CAS is distributed with the Globus Toolkit [15].

2.2.2 GARA: End-to-End Quality of Service for High-end Applications and G-QoSM

GARA [18] represents a modular and extensible architecture for resource reservations to support end-to-end application quality of service (QoS) in Grids. It offers a single interface for reserving different types of resources (network, CPU, disk), and focuses on provisioning generic solutions and algorithms for different types resource managers (the heart of GARA). Reservations (and QoS) are specified through a specialized C-API, composed of client and Globus [54] modules for admission control. GARA is built on three levels: low-level QoS RMs, a QoS component for interfacing with the low-level RMs and provisioning the common interface, and high-level libraries (at the user level) for leveraging reservation synchronizations for user-level applications. The prototype supports only finite reservations, with three main classes of elements: reservations, resources, and QoS elements. All communications client - agreement provider are done through a specific API, and the underlying language for messages is RSL, with only one QoS quantitative parameter per reservation request [18].

G-QoSM [55] builds on top of GARA in order to provide support for applications utilizing Grid computing infrastructure and requiring the simultaneous allocation of resources, such as compute servers, networks, memory, disk storage and other

specialized resources. Collaborative working and visualization are examples of such applications. G-QoSM is a framework for QoS management and allows Grid users to specify, locate and execute Grid jobs with QoS constraints on Grid enabled resources. The framework provides three particular features: 1) support for resource and service discovery based on QoS properties; 2) support for providing QoS guarantees at middleware and network level, and establishing SLAs to enforce these; and 3) providing QoS management on allocated resources based on a pre-negotiated SLA.

2.2.3 SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems

SNAP [19] tries to overcome previous resource managements by providing a generic framework instead of considering specialized classes of resources. The generalized framework maps resource interactions onto a well defined set of platform independent service level agreements. SNAP represents the instantiation of this generalized framework, which provides a management infrastructure for such SLAs. However, the entire approach is generic enough and can be used beyond the Grid domain and Globus Toolkit in particular [19, 56, 57].

The SLAs are categorized as: task service level agreements, resource level agreements, and binding service level agreements. A minimal number of scenarios are also introduced, to provide a basic understanding how SNAP should be used in practice: file transfer service, job staging with transfer service, resource virtualization. Also, for agreements realization, a supporting protocol is introduced that supports the SLA management. [58]

2.2.4 WS-Agreement

WS-Agreement is a specification for resource reservations in grid environments. It is a second version of the OGSI-Agreement, developed in the GGF context. Cremona is a project developed at IBM as a part of the ETTK framework [21, 24, 25], which represents an implementation of the WS-Agreement specification. Its architecture separates multiple layers of agreement management, orthogonal to the agreement management functions: the Agreement Protocol Role Management, the Agreement Service Role Management, and the Strategic Agreement Management. Cremona focuses on advance reservations, automated SLA negotiation and verification, as well as advanced agreement management.

2.2.5 Commercial Workload Management Systems

IBM WebSphere Extended Deployment (XD) is a good example of a commercial workload management system. It offers a dynamic environment for running mixed application types, while also comparing business values for various allocations in order to maximize the business value. Business requirements on the common infrastructure include quantifying and satisfying service level agreements (SLAs) for external customers. This component can and will interact with other services to obtain necessary information. Its primary goal is to interact with the external customer and identify the specific characteristics [137]. At the provider level, a policy-based management component imposes optimal allocations and maintains the integrity of the provider's

environment. The engine examines the policies to determine a configuration with the best "score," which represents the optimal resource configuration [138, 139].

2.2.6 Other Systems

Other groups have pursued similar paths for resource sharing. The most notable ones are the SPHINX framework, developed at the University of Florida, and the Grid Service Broker. SPHINX is policy based scheduling framework for Grid-enabled resource allocations [59]. This framework provides scheduling strategies that (a) control the request assignment to grid resources by adjusting resource usage accounts or request priorities; (b) manage efficiently resources assigning usage quotas to intended users; and (c) supports reservation based grid resource allocation.

The Grid Service Broker, a part of the GridBus Project, mediates access to distributed resources by (a) discovering suitable data sources for a given analysis scenario, (b) suitable computational resources, (c) optimally mapping analysis jobs to resources, (d) deploying and monitoring job execution on selected resources, (e) accessing data from local or remote data source during job execution, and (f) collating and presenting results. The broker supports a declarative and dynamic parametric programming model for creating grid applications [60].

KOALA [37] has been designed and implemented by the PDS group in Delft in the context of the *Virtual Lab for e-Science* (VL-e) project. The main feature of KOALA is its support for co-allocation; that is, the simultaneous allocation of resources in multiple clusters comprising a Grid to a single application consisting of multiple stages. Currently, KOALA supports processor and memory co-allocation and makes use of the

SGE local resource managers. KOALA makes use of the *Close-to-File* policy (CF) or the *Incremental Claiming* policy (IC) for scheduling jobs. The most important feature that enables this work is its capability to perform co-allocations. However, KOALA does not support controlled resource sharing while work is under progress to include an uSLA-based resource allocation similarly to the one proposed in this thesis [48].

2.3. Simulators

Here I describe succinctly several Grid simulators that target similar problems in Grid as GangSim, the tool proposed in this thesis for uSLA-based scheduling studies.

2.3.1 Bricks

Bricks [61] was the first proposed Grid simulator designed to investigate scheduling issues. Its motivations were the proposal and design of an adequate tool for studies and comparisons of scheduling algorithms and frameworks, under various structural and workload conditions, in the objective of providing reproducible results. Bricks scheduling research focus on multi-client, multi-server Grid scenarios. Bricks allows the simulation of various behaviors: resource scheduling algorithms, programming modules for scheduling, network topology of clients and servers in global computing systems, and processing schemes for networks and servers.

It considers the following interacting constituents the global computing system and the scheduling unit: The global computing system consists in clients submitting jobs, servers executing the jobs and the network. Servers and networks are characterized by their performance, workload or congestion, and their variance over time. Servers and

networks are modeled as queuing systems. Jobs are characterized by the size of their parameters/results and the number of computing operations they require. The scheduling unit contains a network monitor measuring network bandwidth and latency, a server monitor measuring performance, load, and availability of servers, a resource data base module storing the all measurements results and serving as a scheduling-specific database, a predictor reading the measured information and predicting resource availability and a scheduler allocating tasks on server based on the resource data base and predictor information [44].

2.3.2 SimGrid

SimGrid [62] is among the most popular simulation tools for Grid research. The main motivation behind the design and development of SimGrid was the necessity of simulation tools to study single-client multi-servers scheduling in the context of complex, distributed, dynamic, heterogeneous environments. Since in its general form, the scheduling problem is NP complete, most of the proposed scheduling algorithms are heuristics. SimGrid provides a set of abstractions and functionalities to build a simulator corresponding to the applications and infrastructures characteristics. In SimGrid resources are modeled by their latency and service rate. These characteristics may be set as constants or evolve according to previously collected traces. The topology is fully configurable. SimGrid considers execution time prediction errors allowing the user to understand the behavior of the scheduling algorithms under complex situations where execution time cannot be accurately predicted [44].

2.3.3 GridSim and ChicSim

Like SimGrid, GridSim [63] is a simulator to investigate scheduling issues in Grids. GridSim was proposed and designed after SimGrid. Its motivations are quite similar. One main difference concerns its focus on Grid economy, where the scheduling involves the notions of producers (resources owners), consumers (end-users) and brokers discovering and allocating resources to users [44].

ChicSim is a modular and extensible discrete event Data Grid simulation system built over Parsec that has been used to evaluate a wide variety of scheduling and replication algorithms [64]. Like GangSim, ChicSim models a Grid as a collection of sites. However, ChicSim does not include notions of VOs or groups and has no support for site usage policies. Similar comments apply to MONARC [65], a simulator developed to evaluate the performance of data processing architectures for physics data analysis, and GridSim, which models various components of distributed systems, but does not address the representation and evaluation of policies.

2.4. Taxonomy of Policies for Access Control

A policy is the statement expressed by one or more owners or administrators of a resource about how the resource can be accessed and used. Policies can be either stored in attribute certificates that extend the authorization mechanism or specialized services that provide on demand these policies. Policies are parsed when required by various inference engines or schedulers, having as result what actions should be performed in response to the resource request.

Authorization Policies: An authorization policy specifies actions that subjects are permitted or prohibited to invoke on managed objects [66]. The entire set of policies that refers to a single resource forms the authorization set that must be enforced, and which must be passed in order to be granted resource access. By using policy statements, an application may address the changing requirements of each resource and session independently. Policies governing resources have been traditionally expressed using access control lists (ACL). This model assumes a centralized control where the owner grants and revokes manually authorizations using lists of rights or other simple associations between names and rights (see previous discussion).

Obligation Policies: In contrast to authorization policies, an obligation policy specifies which actions a subject should (or should not) perform.[66] Subjects can be trusted to perform them, but also monitoring and enforcing mechanisms should be deployed to check how a subject respects obligation policies. Negative obligations are not equivalent with negative authorizations. They act only at the subject part as filters and there are no restrains for the managers to implement them (e.g., instructors must not disclose student evaluations, while students are not forced to implement the same obligation).

Delegation Policies: Authority delegation is essential for automated and scalable operations. It is the way how virtually many modern organizations work. The main authority delegates portions of its authority, portions that become more specific as the delegation chain increases. The person at the end of such a delegation chain has the actual responsibility to act for signing a contract, for operating an equipment, etc. This

authority delegation is traditionally accomplished through a collection of policies and procedures and defines how employees should conduct their activities inside the organization.[66] Secure delegation occurs when one object authorizes another object to perform some task using some of the rights of the initiator. The possibility to verify that an object claims to be acting on another's behalf is somehow a requirement in modern systems. The authorization lasts as long as the target object provides the service or until the delegation authorization expires or is withdrawn.

Roles: Roles are defined as organizational identities composed of rights and duties for an authorized user. A role can be assigned to a single user, to another role, to a group of users or to all users in a system (using special keywords). It is also important to note that roles are not defined in isolation, but in social-like environments. The notion of role does not add any power to a security context, but instead improves manageability by adding an optional level of indirection. A high-level criterion to distinguish roles is to differentiate between elementary roles, just roles, and aggregate roles. In order to define a finer model, it is useful to make distinction also between various types, to define role models (how roles are specified and which relations can be established), administrative models (where the authority for creating and assigning roles resides), assignment models (how roles are assigned), task representations (relation between roles), to define consistency requirements and to specify implementations [66].

2.5. Discussion

Resource access policies typically enforce authorization rules. They specify the privileges of a specific user to *access* a specific resource or resource class, such as submitting a job to a specific site, running a particular application, or accessing a specific file. Resource access policies are typically binary: they either grant or deny access. In contrast, uSLAs as proposed in this dissertation govern the *sharing* of specific resources among multiple groups of users. Once a user is permitted to access a resource via a resource access policy, then other mechanisms can step in to govern *how much* of the resource the user is permitted to consume. Such mechanisms can be QoS rules [18], currencies [60, 63], or uSLAs [67], as introduced in this dissertation for Grid resources.

Resource uSLAs can be sensitive to the *demand* for the resource: the policy may allow a user to use large quantities of a resource in the absence of contention, and lesser quantities in the presence of contention [38, 48], as detailed later in this dissertation.

CHAPTER THREE

USAGE SERVICE LEVEL AGREEMENTS IN GRID ENVIRONMENTS

In the Grid domain [68], I envisage a three-layer structure in which sites provide resources (computers, storage, networks, and high-level services) to virtual organizations (VOs), which in turn provide those resources to their own members. These participants are either resource owners, resource consumers or both. Owners represent the class of participants that provide either outsourcing services or direct access to their computing resources, while consumers represent the class of participants interested in harnessing the aggregate power of shared resources.

In this resource sharing environment, owners and consumers negotiate uSLAs to establish what resources are made available for use by others. Owners want to express (and to enforce) the uSLAs under which resources are made available to consumers. Consumers want to access and to interpret such uSLAs statements to monitor their agreements and to guide their activities. These activities involve the allocation of aggregate resources provided by different owners for different internal purposes, and to orchestrate distributed computation to use those aggregated resources efficiently. Both owners and consumers want to verify that these agreements are applied correctly.

Owners and consumers may be nested: an owner may function as a middleman, providing access to resources to which the owner has itself been granted access by some other owners. Thus, uSLAs issues can arise at multiple levels in such scenarios.

The main question addressed is: "*How should uSLAs be represented and handled in Grid environments in order to act as an organizing principle for resource management?*" In addressing this question, I build on previous work concerning the specification of local resource scheduling policies[59, 69-73]; the negotiation of SLAs with remote resource sites [19, 21, 22, 39], and the expression and management of VO usage policies [38, 59, 74].

3.1. Scenarios Revisited

In Chapter 1, I provide an introductory scenario for scientific collaborations that vary widely in size and scale. In this chapter, I present concrete details based on Grid3/OSG. In addition, I discuss a scenario for a large company acting in several countries (and in several markets) and one for a high level service from an external provider. In each scenario, some common uSLA examples are also provided.

3.1.1. Grid3 Scenario

In the first scenario, the Grid3/OSG comprises tens of institutions and hundreds to thousands of individual investigators that collectively control thousands of computers and associated storage systems [31, 75]. Each individual investigator and institution participates in, and contributes resources to multiple collaborative projects that vary in scale and formality. Figure 3.1 depicts a graphical representation of the Grid3 sites.

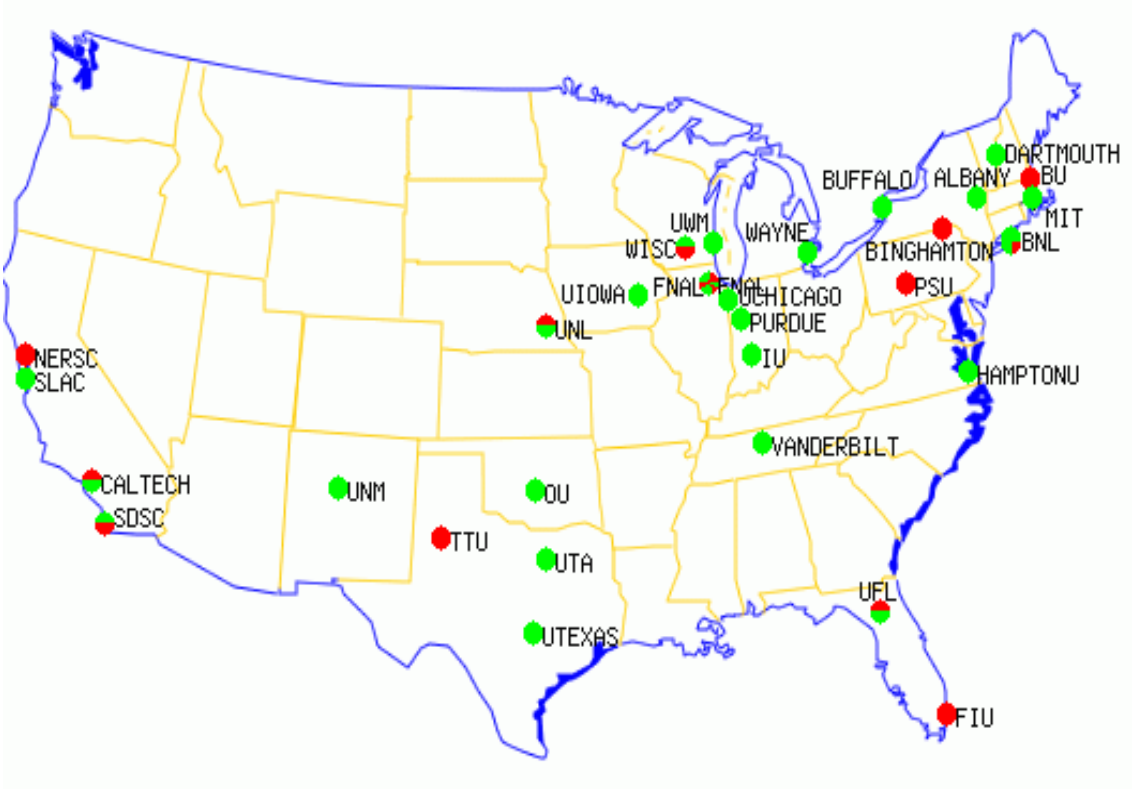


Figure 3.1: Grid3 Sites and Instantaneous Utilizations - The Grid Catalog Monitoring System (GridCat) snapshot

In this environment, several VOs exist that are composed of users with various common interests and applications. The most common ones are the USATLAS [76], Sloan Digital Sky Survey (SDSS) [77] and iVDGL [78] VOs (see Figure 3.2). USATLAS users simulate the collisions of protons on protons at 14 TeV at the LHC for the CMS experiment – applications are composed of hundreds of embarrassingly parallel programs with large input/output files. The SDSS VO users measure the distance to, and the masses of, clusters of galaxies in the SDSS data set – applications

are composed again of many components, but in this case they have input/output dependencies [79] that can be represented using direct acyclic graphs (DAGs). The iVDGL VO performs protein sequence comparisons at increasingly larger scales. This application uses various size workflows in which a single BLAST job has an execution time of about an hour - the exact duration depends on the CPU, reads about 10-33 kilobytes of input, and generates about 0.7-1.5 megabytes of output.

Grid3 sites are sponsored either by different VOs or directly by the hosting institutions. Each site has usage policies (expressed sometimes as "provide 30% of resources to USATLAS") that are enforced by means of a local resource manager (RM). The sites we used for local usage policy analysis are located at Lawrence Berkeley National Laboratory (PDSF), the University of Buffalo (UBuffalo), the University of Chicago (UChicago) and the University of Milwaukee (UMN). At PSDF, the RM is LSF; at UChicago the RM is Condor; at UMN the RM is PBSPro, and at UBuffalo is Maui with PBS. On each of these sites, the Grid3 settings are as follows: all users from a VO are mapped to a local common ID, and the local RM is either configured to provide that local ID some predefined quantities or a specific fair-share priority relative to other users. With so many different resource managers, it is important to express them in a common language.

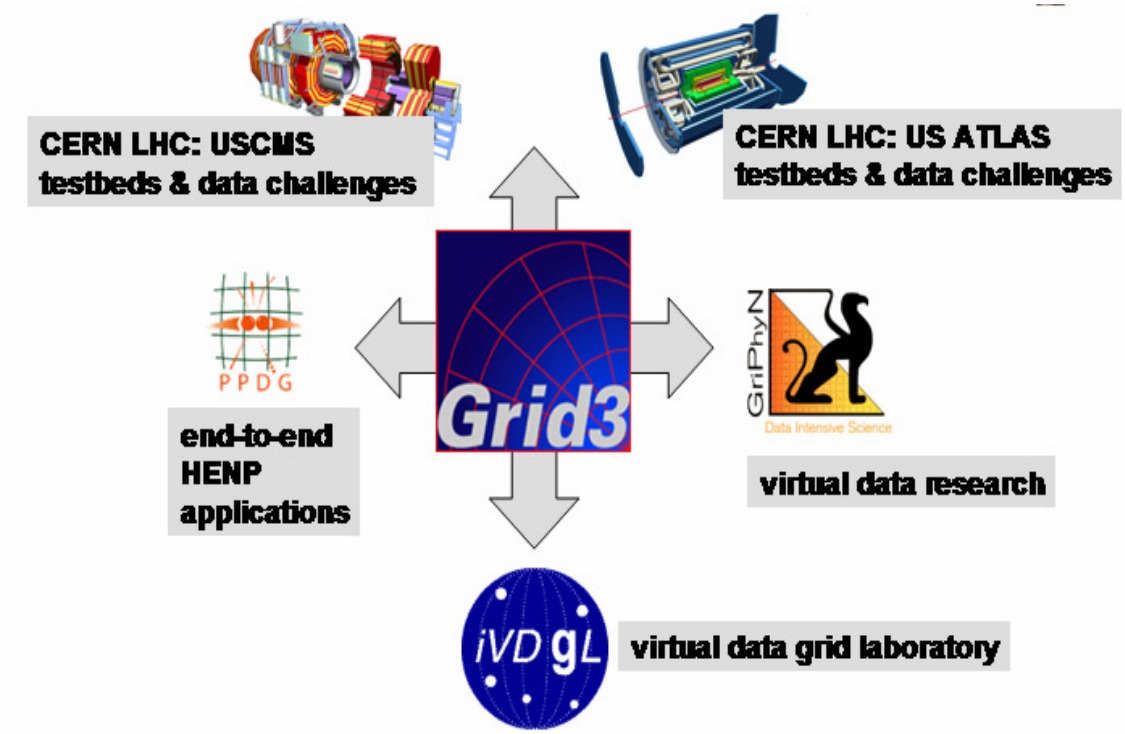


Figure 3.2: Virtual Organizations Operating on Grid3

For the Grid3 scenario, some of the requirements include the provisioning of fair share allocation policies capable of expressing situations both with and without contention. Usually, resource providers (universities and laboratories) and resource consumers (scientists from different domains) want access to these resources pooled together according to various needs. For example, before important conferences we have observed that Grid utilization increases and higher job contention occurs, while during holidays most resources are free for long time intervals [80]. These observations presented by Iosup et al. [80] motivate our introduction of a uSLA that ensures

"whenever there is no contention users can use as many resources as possible, while when contention occurs, the resources are allocated according to pre-defined rules that provide the incentives for Grid participation." The following sharing example is widely accepted by each individual site (or with different variations in terms of the amount of resources provided) [81]: "*there are three types of incoming jobs to balance: one from CMS, one from ATLAS, and one from iVDGL. We call them USCMS-Prod, USATLAS-Prod, and IVDGL. We want USCMS-Prod and USATLAS-Prod to get an equal share of available CPUs, but IVDGL should get a small fraction, of the resources, if there is contention (a 4th of what the others get)."* Figure 3.3 shows this controlled resource sharing scenario for the University of Chicago resources.

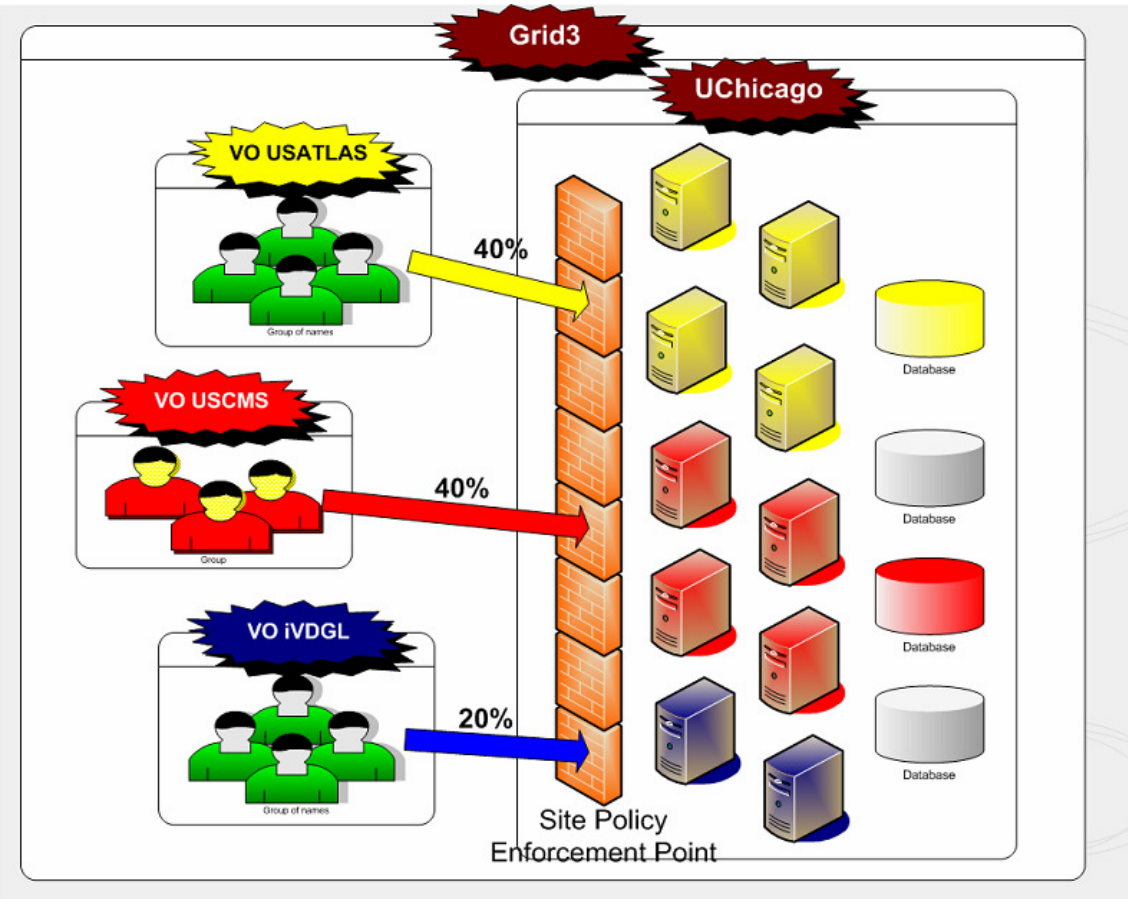


Figure 3.3: Graphical View of the UChicago Resource Sharing

3.1.2. Multi-Market Company Scenario

The second scenario I consider comes from the business world. In this case, a company active in multiple financial markets owns computational resources in each country where it operates [23]. These resources are used primarily for trading screens (see Figure 3.4). Since traders are active only during the day, the company allows other business-related applications (e.g., portfolio risk evaluation - a compute and data intensive application; market prediction - another compute and data intensive application) to run on these resources at night instead of deploying an additional specialized computing farm. The company has two usage policies: *the pooled resources must be used according to each local market's usage policies* and *the computing nodes must be available to traders from 7:00 to 22:00, during the local trading day.*

At the company level, the aggregated resources can be further sub-allocated by means of company wide uSLAs based on the current importance of each business application. For example, in our scenario the *market prediction* application might be more important, thus the company provides *70% of the aggregated resources from each individual market for its computations whenever resources are not used by local users* (which becomes a VO level uSLA).

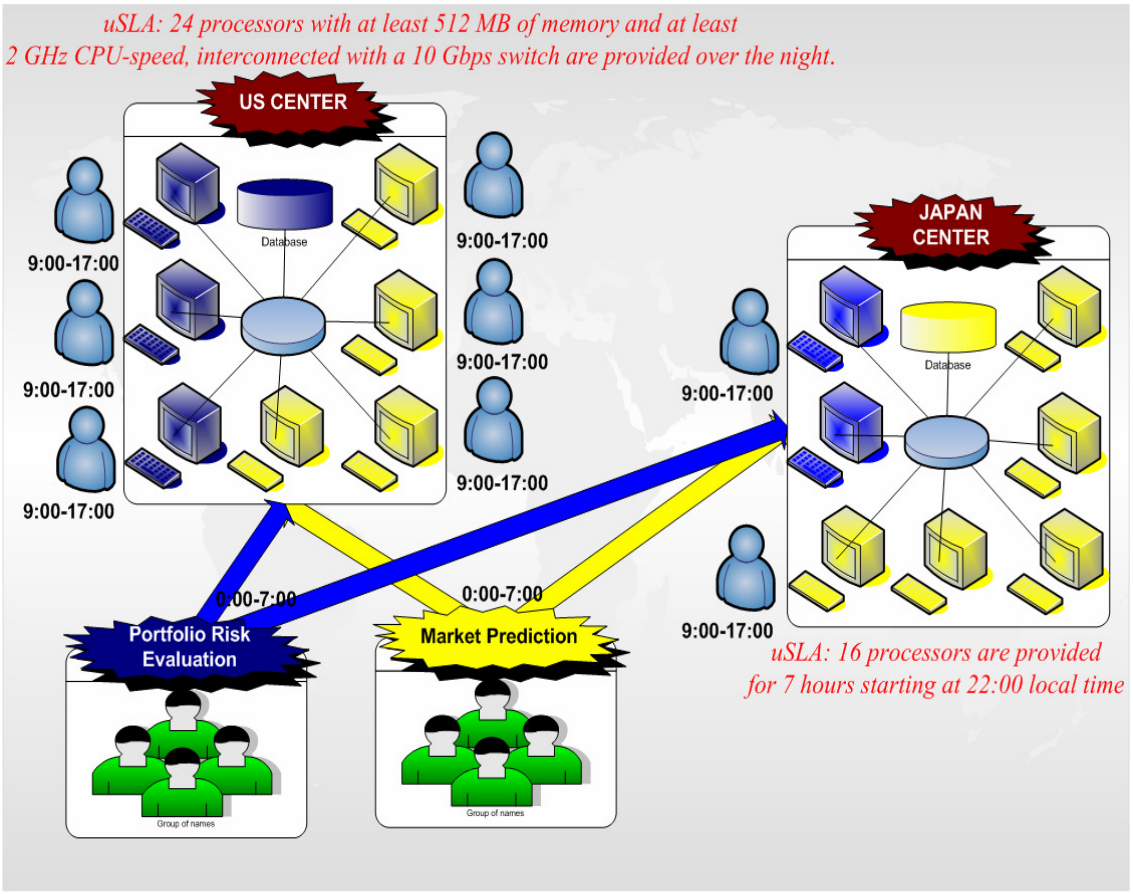


Figure 3.4: Graphical View of Multi-Market Company Resource Utilization

Each local center also imposes its own local policies. At the lowest level, such usage policies include detailed resource sharing information (they describe the physical resources to be shared): "*16 out of 32 processors are provided on cluster X for 9 hours starting at 22:00 local time*" or "*24 machines with at least 512 MB of memory and at least 2 GHz CPU-speed, interconnected with a 10 Gb/s switch are provided from 22:00 to 7:00.*" The second example captures a multi-resource uSLA, which are important in complex distributed systems.

3.1.3. Outsourcing Scenario

In the third scenario, I envisage that a community out-sources some high level services to reduce deployment and operational costs. Community members acquire resources and services from independent utility providers that specialize in providing those services (see Figure 3.5). Service examples include scheduling prediction services, monitoring services (MonaLisa [65]), or community authorization services (e.g., DOE certificate authority [82]).

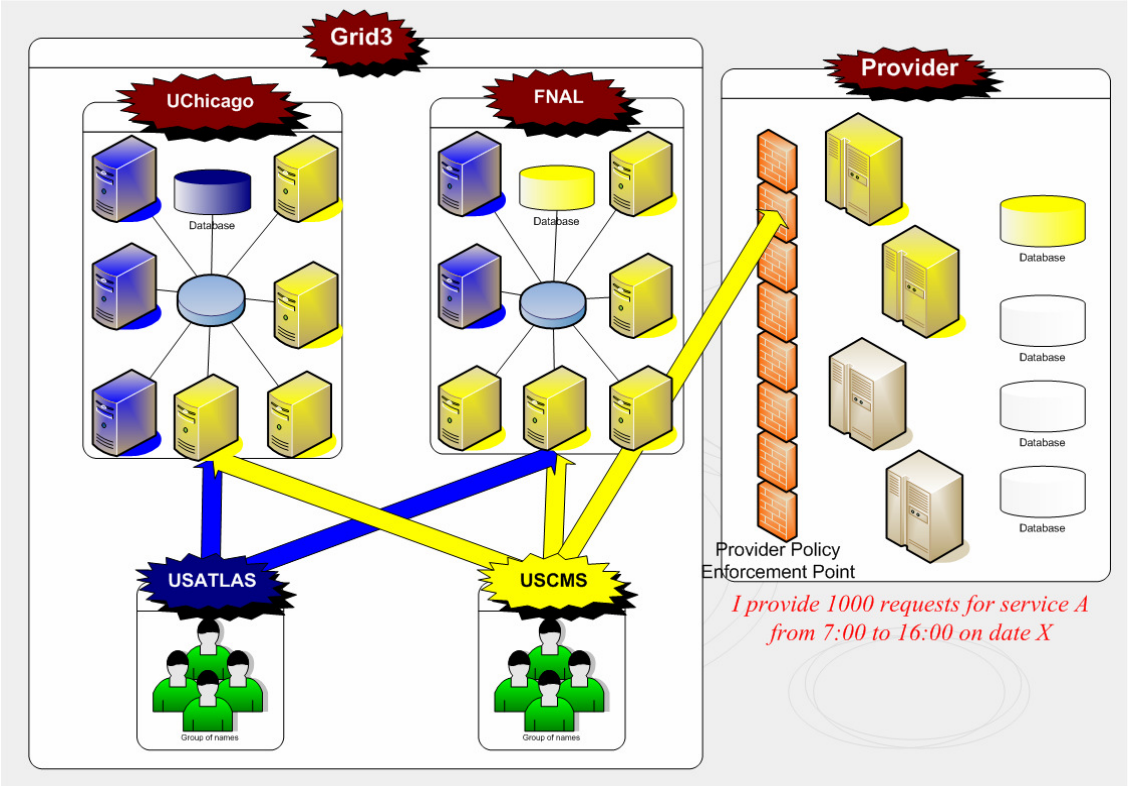


Figure 3.5: Service Outsourcing Scenario

In this scenario, the service provider requires usage policies to express the amount of resources or services he is willing to provide: *I provide 1000 requests for service A from 7:00 to 16:00 for 1 month for any remote user from Grid3* or *I accept 1000 requests for service A from 7:00 to 16:00 on date X for any remote user from Grid3* [83].

3.1.4. Requirements for uSLAs

Usage policies in all the above scenarios provide support for controlled sharing from raw resources (CPU, disk space or network bandwidth) to complex services (such as various Grid-level services). They express simple usage conditions *("30% for a month for consumer A")* to complex ones (see previous examples). These uSLAs affect only specific sets of resources. For example, site-level or center-level requests must affect only local resources, while VO-level or company-level resources affect aggregated virtual resources and, thus, the site resources pooled together.

The Grid3 scenario requires support for simple uSLAs that any current cluster management tool enforces at the site level, but it is important to insure that Grid-level schedulers and brokers have access to these uSLAs to achieve higher resource usage. At the VO level, similar uSLA support is required to ensure that Grid entities get a certain percent of the aggregated resources when there is contention. The multi-market company scenario represents a different case, a business environment where resources are allocated to various activities and group functions based on the business rewards these activities or entities generate. Resources might not be completely allocated for other purposes even if they are available, thus the uSLA must capture such situations.

In this scenario, uSLAs are dependent also on time. I do not consider an additional mechanism for time, because it can be incorporated into the uSLA language specification as a normal precondition. Now, for the last scenario, the out-sourcing company, no additional mechanisms are required, but it represents an interesting study case from a different angle: to show that uSLAs can be used with success for sharing controlled higher-level resource (services) as well as raw-resources (CPU, network, disk).

In the above scenarios, when multiple concurrent requests are present and sufficient resources are not available, contention cases must be handled. This process is referred to as *arbitration* or conflict resolution [84]. The uSLA mechanisms must distinguish among various situations that can occur in practice and treat them accordingly. Usually, in a non-contention case, resources can be acquired as a function of availability and the uSLA specification, while in the contention case several sub-cases can occur.

As the examples make clear, both producers and consumers need to specify various types of uSLAs using an agreed upon syntax and semantics. To support such uSLAs, a system must provide the mechanisms for:

1. Monitoring of utilization and provisioning to ensure that agreements are honored. These mechanisms represent the supporting layer for uSLAs: various guarantees are expressed in terms of how many resources are provided and used, thus correct monitoring can ensure the expressiveness of the uSLAs.

2. Expressing uSLAs for situations both with and without contention and providing clear semantics for each ensures that both consumers and providers can establish well defined agreements upon which resources are used.
3. Identifying legal users to ensure users and resource owners' identities and to identify legal users, uSLA makers and requests. For these mechanisms, we rely on the Grid technologies that provide support for authentication and authorization inside large distributed infrastructures [54, 85, 86].

We provide support for monitoring by introducing notions of and mechanisms for aggregated consumer-based monitoring at the raw or abstract resource provider-level (for example, VO-level monitoring at the site level in Grids; fair share rules by specifying the semantics of four possible levels of sharing [33, 87]; and finally, a model for identifying legal users in large and dynamic environments [33].

3.2. Monitoring Approach

Adequate monitoring is important if we want our uSLA mechanisms to be successful. Thus, I developed mechanisms for measuring how resources are used by each resource consumer (VOs and groups) and by the Grid, overall. My goals were to provide mechanisms to monitor Grid-level resource activity, utilization, and performance; to provide VO-level resource activity and resource utilization monitoring; to create customized views of monitoring data including hardware resources (clusters, sites, and Grids) and VO and group usage (in terms of the number of jobs and their characteristics) and workflow types. Ultimately, the monitoring techniques must

provide uSLA-driven support for Grid-resource scheduling. The mechanisms also need to support a large number of resources and uSLA-enabled metrics.

I have both identified key requirements for monitoring and provided solutions for incorporating these new requirements into existing monitoring infrastructures (such as Ganglia [88] and the VO-Ganglia extension [67], GRUBER-SiteMonitor [43], MonALISA [65] and ARESRAN [83]).

3.2.1. Monitoring Requirements

A joint project among GriPhyN, iVDGL, and US ATLAS focused on "Grid-Level" monitoring for research into uSLA-driven scheduling and usage [31]. During this work, we identified several monitoring requirements:

1. Grid-enabled tagging of monitoring data for:
    a. Resource consumer-oriented monitoring data collection, and
    b. Scheduling characteristics and allocation rules collection and publishing;
2. Monitoring data flow: from consumers to uSLA brokering systems, workflow steering mechanisms, or any other automated agents, and
3. Monitoring data aggregation at various levels (raw resources / abstract resources);

In addition to these monitoring mechanisms, our system must also support complex queries, such queries that involved scaling (current time + 2:00) or composition (used CPUs / total number of CPUs).

3.2.2. Implementation Approaches and Technical Solutions

We integrated VO components within the Ganglia monitoring software [31, 67]. Initially, I extended the Ganglia monitoring toolkit to meet the above requirements (the so called VO-Ganglia) and later migrated these mechanisms to other systems, most notably MonALISA and ACDC Monitoring Dashboard. The final technical solution required supporting mechanisms for:

1. Added support for site, VO and group enabled monitoring:

a. Grid-enabled monitoring data collection and local scheduler type, characteristics and allocation rules publishing. For example, in the Grid3 scenario "*configuration knobs*", "*allocations per VO*", "*usages per VO*" were added to the list of monitored characteristics.

b. Introduced provisioning of detailed utilizations and allocations;

2. Introduced support for monitoring data aggregation at various levels / finer information about sites configurations;
3. Added support for automated querying of monitoring information and integration with the uSLA system developed in this dissertation (the GRUBER Grid broker);
4. Introduced support for advanced querying mechanisms to provide the necessary monitoring data used by the GRUBER broker (e.g., GRUBER-SiteMonitor).
5. Introduced support for automated translation from raw quantities to percentages (%) and common descriptions as required for the uSLA monitored metrics.

My technical infrastructure is composed of host sensor collectors, summation meta-daemons and cluster/host automated query interfaces. The infrastructure is able to collect information about hardware usage, VO-related usage, and uSLAs [33]. The architecture and a sample display are shown in Figure 3.6 and Figure 3.7, respectively.

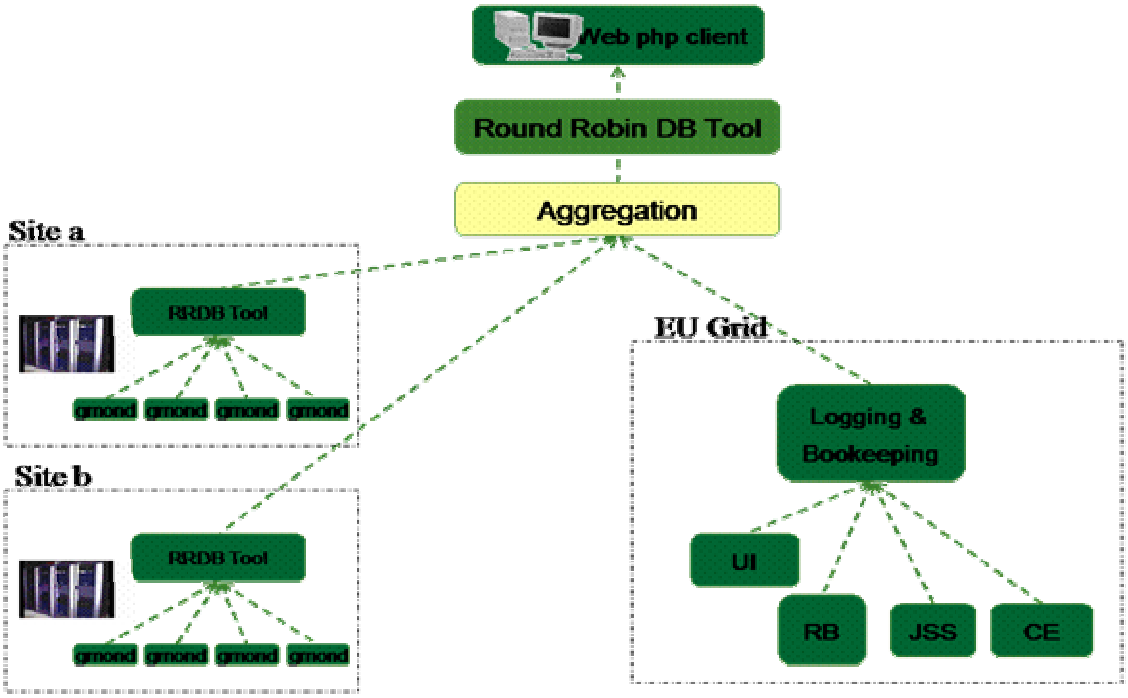


Figure 3.6: VO-Ganglia Prototype

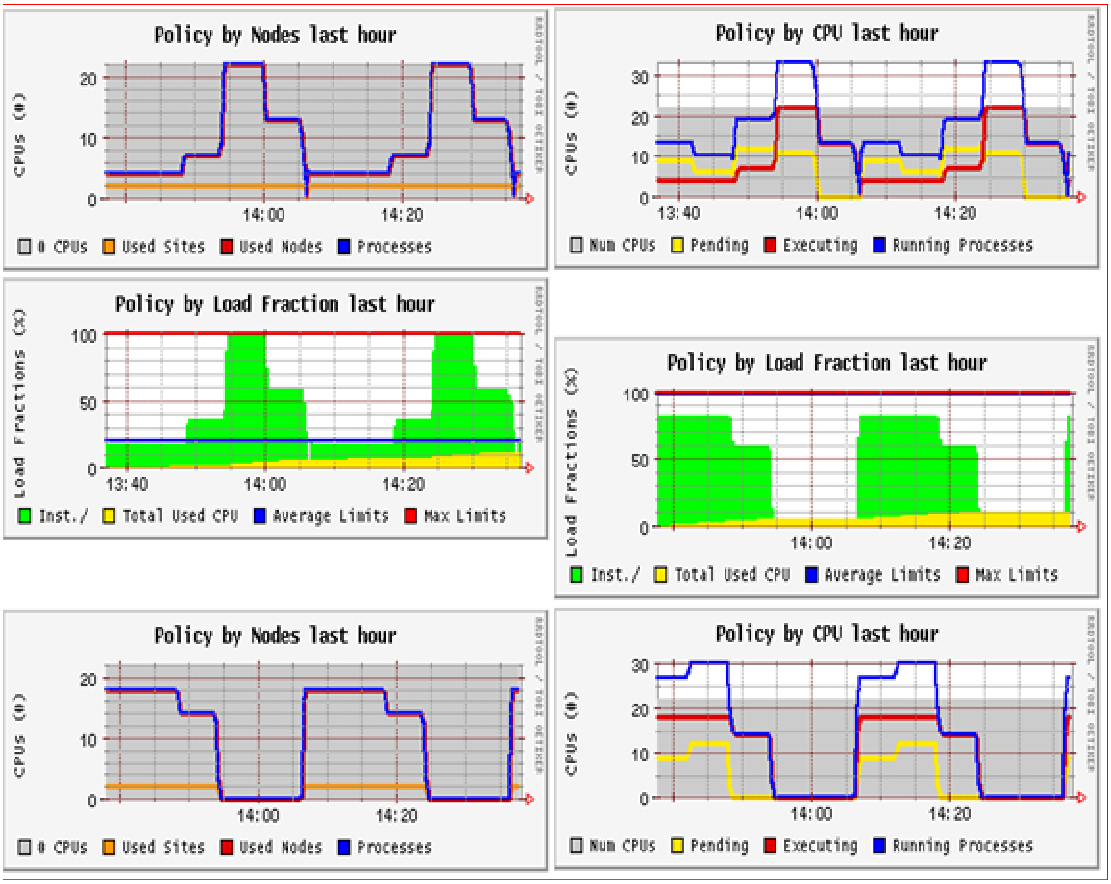


Figure 3.7: VO-Ganglia Reporting Example

3.3. uSLA Semantics

As already stated, uSLAs [29, 38, 39] express a relationship between a resource provider and a resource consumer, and represent provider statements about a contract that governs how its resources are to be allocated to a specific resource consumer. However, without a well-established semantic and syntactic foundation for these uSLAs, their interpretation may become misleading.

I focus in the rest of this section on my proposed semantics and syntax in the Grid context. I have applied this method for specifying uSLAs with success in the OSG/Grid3 context [43, 89-91]. I consider semantics for specifying CPU usage constraints and requests and give examples for the proposed schemas for the three scenarios introduced above. I also present approaches for controlled resources of disk space and higher level services. My design of uSLA semantics matches the requirements identified before, and supports ease of uSLA expression, automated discovery and management of these uSLAs, and multiple resource types. Complete examples for expressing different uSLAs are provided in the next Section.

3.3.1. CPU Resource Support

The uSLA semantics proposed in this section express how controlled resource sharing is performed in the scenarios described in section 3.1. They are generic enough to be also applied at the VO and VO group levels. Next, I provide an algorithmic description about when and how new requests should be admitted in each of the four cases. The

proposed semantics are named after their goals: *no-limit*, *fixed-limit*, *extensible-limit*, and *commitment-limit* [43, 89-91].

The ***no-limit*** uSLA is a statement that specifies no limit. Resources are acquired on a first come first executed basis.

The ***fixed-limit*** uSLA specifies a hard upper limit on the fraction of resources $\mathbf{R_i}$ available to a $\mathbf{VO_i}$. A request to run a job is granted if this limit is not exceeded, and rejected otherwise. More precisely, a job requiring **J** resources is admitted if and only if $\mathbf{C_i + J \leq R_i}$, where $\mathbf{C_i}$ denotes the resources currently consumed by $\mathbf{VO_i}$ at the site. Note that an admitted job will always be able to run immediately, unless the resource owner oversubscribes resources, i.e., $\mathbf{\Sigma_i R_i > 1}$.

The ***extensible-limit*** uSLA also specifies an upper limit, but this limit is enforced only under contention. Thus, under this SLA a job requiring **J** resources is admitted if $\mathbf{C_i + J \leq R_i}$ **or** $\mathbf{\leq C_{free}}$, where $\mathbf{C_i}$ and $\mathbf{R_i}$ have the same meaning as before, and $\mathbf{C_{free}}$ denotes the site's current unused resources. Note that because this policy allows VOs to consume more than their allocated resources, whether or not an admitted job can run immediately may depend on the site's preemption policy.

While the fixed or extensible uSLA are sufficiently expressive for the Grid3 scenarios, their limitations become obvious when moving to more complex ones. For example, the multi-market company case introduces the need for ways to differentiate among usage policies at various levels (center-wide or company-wide ones) and to support different usage policies for different times of day.

Our last uSLA, the ***Commitment-limit*** SLA supports these more complex queries. It specifies two upper limits, an epoch limit $\mathbf{R_{epoch}}$ and a burst limit $\mathbf{R_{burst}}$, and specifies for each an associated interval, $\mathbf{T_{epoch}}$ and $\mathbf{T_{burst}}$ respectively. A job is admitted if and only if (a) the average resource utilization for its VO is less than the corresponding $\mathbf{R_{epoch}}$ over the preceding $\mathbf{T_{epoch}}$, or (b) there are idle nodes and the average resource utilization for the VO is less than $\mathbf{R_{burst}}$ over the preceding $\mathbf{T_{burst}}$. Both periods are modeled here as recurring within fixed time slots. A provider may grant requests above the epochal allocation if sufficient resources are available, but these resources can be preempted if other parties with appropriate allocations request those resources at a later stage. More precisely, any job accepted by the following algorithm is admitted, with the following definitions:

R_{epoch} = Epoch Usage Policy for VOi

R_{burst} = Burst Usage Policy for VOi

BA_i = Burst Resource Usage for VO_i

EA_i = Epoch Resource Usage for VO_i

TOTAL = upper limit allocation on the site

procedure *commitment-AP*

returns accept/reject

Case 1: site over-used by VO_i

1. **if** $EA_i > R_{epoch}$ **then**

2. reject job from VO_i

Case 2: sub-allocated site

3. **else if** $\Sigma_k(BA_k) + J < TOTAL$ **and** $BA_i + J < R_{burst}$ **then**

4. run job from VO_i

Case 3: over-allocated site

5. **else if** $\Sigma_k(BA_k) = TOTAL$ **and** $BA_i + J < R_{epoch}$ **then**

6. schedule job from VO_i

7. **else**

8. reject job from VO_i

Thus, for simplification purposes, I have considered an instantaneous limit (*burst*) as well as a long term limit (*epoch*) that capture how resources are allocated over two time intervals. This uSLA can be extended further by introducing an unlimited number of sharing intervals, which makes it generic enough to express any requirements in practice. I note that the fixed limit and extensible limit can be expressed as particular cases of the commitment uSLA. I work on enriching this set of policies [92], but they represent a minimal set of solutions that meet our requirements and can be implemented easily by means of current site resource managers.

I exemplify next for each of the three scenarios from Section 3.1 how the above uSLAs should be used. The OSG/Grid3 sample policies can be expressed by the extensible uSLA. A site shares its resources among three VOs and allows any of them to acquire more resources when there is no contention. When there is contention, the exact

allocations in this case are 40% for USATLAS, 40% for USCMS and 20% for iVDGL. Also, the specification is instantaneous, in the sense that at any moment this sharing policy should be respected.

In the second scenario, the first company rule which states that local rules should be taken in account, does not need to be specified because it is performed automatically. The second rule can be translated into an extensible uSLA, where all resources are provided 100% to traders. The third rule can be translated as a commitment uSLA, under which at any moment in time 70% of the company resources are provided for remote users' usage, while in the long term there is no guarantee.

In the final scenario, the allowed number of requests must be translated into percentages to be expressed as allocations. Such rules can be expressed easily as fixed uSLAs with fixed time constraints that are captured by the uSLA syntax.

3.3.2. Disk Space Support

Disk space management introduces additional complexity to job management [43]. If an entitled-to-resource job becomes available, it is usually possible to delay scheduling of other jobs, or to preempt them if they are already running. In contrast, a file that has been staged to a site cannot be "delayed," it can only be deleted. Yet deleting a file that has been staged for a job can result in livelock, if a job's files are deleted repeatedly before the job runs. As a concrete example, a site can become heavily loaded with one VO's jobs which cause other jobs to be held in the local queue awaiting their turn. But these waiting jobs do not stop the submission of more jobs. As a result, there may be lots of other input data on the site and the disk space used will continue to grow. On the other

hand, some jobs are not getting a turn to finish and delete their files. If the rate of input data being copied to the site is higher than the rate of job completion, then the disk space will fill.

Based on the UNIX quota system, the same four uSLAs can be implemented with success. However, in this case once a file is saved at a site and the allocation is higher than the uSLA limit allows (extensible-limit and commitment limit cases), the space cannot be preempted without evicting the violating files to other sites. Our approach builds on the UNIX quota system, which prevents one user on a static basis from using more than his hard limit (but it still considers soft and hard limits similarly to the commitment limit). More precisely, for scheduling decisions a list of site candidates that are available for use by a VO_i for a job with disk requirements J is built by executing the following logic, with the following definitions:

```
S          = Site Set
k          = index for any VO != VOi
IPi        = Epoch uSLA for VOi
ISPi            = Instantaneous uSLA for VOi
IAi        = Instantaneous Resource Usage for VOi
TOTAL      = upper limit allocation on the site
```

```
procedure commitment-AP_disk
        returns S (list of available sites)
```

```
1. for each site s in site list G do
   # Case 1: over hard-limit site by VO_i
3.    if IA_i > IP_i for VO i at site s
4.      next
   # Case 2: over soft-limit site by VO_i
5.     if IA_i > ISP_i and time < grace period for VO i at
site s
6.      if Σ_k(IA_k) < s.TOTAL – J && IA_i + J < IP_i then
7.        add (s, S)
8.      next
# Case 3: un-allocated site
9.    else
10.      if Σ_k(IA_k) < s.TOTAL – J && IA_i + J < IP_i then
11.        add (s, S)
12.      next
13. return S
```

3.3.3. Higher-Level Services Support

The final type of resource I consider is a Grid service (or s high-level resource). Again, I use the same uSLAs as in the CPU case to encode resource availability from a provider's point of view [93]. However, Grid services are difficult to quantify in term of their utilization - a weather service and a matrix multiplication service are difficult to compare in terms of resource consumption without a through service performance model analysis. Thus, there is no uniform way for expressing how much computing power a certain service may require to serve a certain request. To overcome this complexity and provide a simple solution to this problem, I maintain the CPU semantics unchanged for the uSLAs, but with different utilization metrics; thus, instead of CPU utilization, the number of requests a client can perform on a certain service is considered for the uSLA algorithms. While this approach may seem an over-simplification, the end result is similar: a service provider states by means of the uSLAs how many requests a certain consumer can perform on its resources. Based on this approach, the algorithm to supports controlled Grid service sharing is identical with the one for controlled CPU sharing.

However, we present next a variation that allows advance service reservations (a request can be made well in advance of its starting time). It applies a pre-specified uSLA on each "future" sub-interval that results from a variation in terms of either service requests or allocations. The algorithm for accepting new advance reservations is introduced next, where:

```
   R           = resource request
   A_j         = allocated resources
   R_j         = requested resources
   Allocations = set of accepted allocations
   Requests    = set of already accepted requests

procedure request_check (R) returns response
1. response = true
   # Stores availability on the considered interval
2. S = empty
   # Identify all requests overlapping current request
   # (save their start/end times and requested quantities)
3. foreach R_j in Requests do
4.   if R_j time overlaps R time then
5.        save S, R_j.start, + R_j.attributes
6.        save S, R_j.stop, - R_i.attributes
7.   fi
8. done
   # Identify all allocations overlapping current request
   #  (update accordingly previous values with these
allocations)
```

```
9. foreach Aj in Allocations do
10.  foreach Tj in S do
11.       if Aj overlaps Tj then
12.          save S, Tj, + Aj.attributes
13.       fi
14.  done
15.  done
   # Check constraints (available resources)
16.  compute Request = R.attributes
17.  foreach Availability in S do
   # Apply the appropiate uSLA algorithm (hard limit
example here)
18.       response     =     uSLA_fixed-limit     (Request,
   Availability)
19.  done
20.  if  response == true then
21.       update accordingly one of the overlapping uSLAs
22.       add R to Requests
23.  fi
24.  return response
```

In this simple way, the controlled resource sharing mechanisms devised in this chapter can be applied with success not only to raw resources but also to service-like

types of resources and also for advance service reservations. I must also note that such a service manager is already available for Grid services (e.g., ARESRAN or SAML).

3.4. uSLA Syntax

I have considered two syntaxes for uSLAs: a simpler one based on allocations and one based on WSLA. My starting point for the first approach is the Maui [94] syntax for specifying allocations. Maui supports three types of fair share limits: "*at least limit,*" "*average limit*" and "*at most specification.*" In the first case, when the utilization for a group goes below the limit, the mechanism increases the priority of the jobs from that group (expressed as a real number preceded by a "+"). In the second case, whenever the utilization for a group is different from the limit, the fair share mechanism either increases the priority (for under-utilization) or decreases the priority (for over-utilization) of jobs from that group. In the final case, when the utilization for a group goes above the limit, the fair share mechanism decreases the priority of jobs from that group (expressed as a real number preceded by a "-"). Our first syntactic form is represented as a set of *allocations* of the form:

<resource-type, provider, consumer, start, epoch-allocation, burst-allocation>

where:

resource-type ::= [CPU | NET | STORAGE]

provider ::= [site-name | vo-name]

consumer ::= [vo-name | (vo-name, group-name) | ANY]

start ::= date-time | time | *

epoch-allocation ::= (interval | *, percentage) | -

burst-allocation ::= (interval | *, percentage) | -

ANY ::= *matches any name*

* ::= *means instantaneous*

- ::= *means not specified*

To show the capacity of the above syntax, I show next how it can be used for the three scenarios introduced before. The uSLAs of the OSG/Grid3 environment are expressed under this syntax as a set of three rules with values only for the instantaneous allocations (where the burst period is considered a 5 seconds interval):

<CPU, UChicago, USATLAS-Prod, *, -, (5, +40) >

<CPU, UChicago, USCMS-Prod, *, -, (5, +40)>

<CPU, UChicago, iVDGL, *, -, (5, +20)>

In the second case multi-market scenario, I distinguish between two types of rules: company-wide rules and center-wide ones – a simple analogy could be drawn between the site and VO level as proposed by Foster et al. [4]. The company-wide uSLAs are expressed under our first syntax as a set of two rules:

```
<CPU, Company, ANY, 22:00, (9*3600, 100), (5, 100)>
<CPU, Company, Prediction, 22:00, (9*3600, -70), (5, -70)>
```

while, at the center level, the first rule is expressed as (after translating the 16 CPU power into a percentage of the total available computing power):

```
<CPU, Company, ANY, 22:00, (9*3600, 50), (5, 50)>
```

However, this syntax has its limitations for expressing sharing rules about resources of different types. First, this syntax does not provide a mechanism for specifying monitoring requirements. Second, it does not support the specification of complex conditions (AND, OR, etc.).

Thus, I propose a uSLA syntax based on the WS-Agreement specification, to take advantage of its high-level structure SLA specification and of available parsers. The objective of a WS-Agreement specification is to provide standard means for establishing and monitoring service agreements. The specification draft comprises three

major elements: a description format for agreement templates, a basic protocol for establishing agreements, and an interface for monitoring agreements at runtime.

For the uSLA syntax, I use a schema that includes from the WS-Agreement specification support for resource monitoring metrics and goal specifications [21, 24, 43]. The resource monitoring metrics describe how various utilizations must be measured or how these quantities should be collected from an underlying monitoring system. A goal specification provides support for describing the targeted allocations in a form that can be parsed by automated agents. The other elements (i.e., obligations and handlings violation) were considered beyond the scope and capacity of current site and VO resource managers.

The schema of this grammar is described next. First, the monitoring metric element defines how a certain resource metric required for a guarantee should be measured.

```
   <!-- MonitoredMetric -->
<xsd:complexType name="MonitoredMetricType">
   <xsd:attribute name="name" type="xsd:string" />
   <xsd:attribute name="method" type="xsd:string" />
   <xsd:attribute name="type" type="xsd:string" />
   <xsd:attribute name="interval" type="xsd:integer" />
   <xsd:attribute name="notification" type="xsd:boolean"
/>
</xsd:complexType>
```

The next element of the grammar, MonitoredType, describes the entire list of monitored metrics required in enabling the considered uSLA. This list can have zero or more of required metrics that must be monitored.

```
   <!-- Monitored Type -->
<xsd:complexType name="MonitoredType">
  <xsd:sequence>
     <xsd:element name="MonitoredMetric"
           type="MonitoredMetricType" minOccurs="0" />
   </xsd:sequence>
      <xsd:attribute name="name"
                type="xsd:string" use="optional" />
</xsd:complexType>
```

The precondition element identifies of the entity for which the uSLA is defined and the name of the provider:

```
   <!-- Precondition -->
<xsd:complexType name="PreconditionType">
  <xsd:sequence>
    <xsd:element name="consumer"
```

```
            type="xsd:string" minOccurs="0" />
    <xsd:element name="provider"
                type="xsd:string" minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>
```

The goal element describes the conditions under which a resource is provided. It uses the constraint element for defining conditions (with *LessEqual*, *Equal* and *GreaterEqual* corresponding to the semantics introduced earlier for -, <*space*>, + signs, while *Range* has a special meaning for time specifications):

```
   <!-- Goal -->
<xsd:complexType name="GoalType">
  <xsd:sequence>
     <xsd:element type="ConstraintType" minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>

   <!-- Constraint -->
<xsd:complexType name="ConstraintType">
   <xsd:attribute name="type" type="xsd:string"
```

```
      values="LessEqual, Equal, GreaterEqual, Range" />
   <xsd:element name="Metric" type="xsd:string" />
   <xsd:element name="Value" type="xsd:literal" />
</xsd:complexType>
```

Similarly to a MonitoringType element, a GuaranteeElement contains a list of all guarantees that a resource provider agrees to when providing the resources.

```
   <!-- Guarantee Type -->
<xsd:complexType name="GuaranteeType">
   <xsd:sequence>
       <xsd:element name="Precondition"
type="PreconditionType"
               minOccurs="0" maxOccurs="1" />
       <xsd:element name="Goal" type="GoalType"
minOccurs="0"
               maxOccurs="1" />
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string"
use="required" />
</xsd:complexType>
```

The final element of the grammar is the uSLA element, which is composed of several monitored and guarantee elements.

```
   <!-- usage SLA -->
<xsd:complexType name="uSLA">
    <xsd:attribute name="Monitored" type="MonitoredType"
                  minOccurs="1" />
    <xsd:attribute name="Guarantee" type="GuaranteeType"
                  minOccurs="1" />
    <xsd:attribute name="name" type="xsd:string"
             use="required" />
</xsd:complexType>
```

Now, I show next how it can be used for the three scenarios introduced before. The first OSG/Grid3 example is represented using this syntax as follows. Three metrics are monitored (by means of MonaLisa, for example, and these values are retrieved from a certain URL): *CPUBurst-Met-USATLAS*, *CPUBurst-Met-USCMS* and *CPUBurst-Met-iVDGL*.

```
<uSLA name=”Grid3 uSLA (Scenario 1)”>

    <!-- Define Monitored Metrics (and acquisition
mechanism) -->
    <Monitored>
    <MonitoredMetric name="CPUBurst-Met-USATLAS"
        method="http://URL/CPU?vo=USATLAS&t=5"
        interval="5" type="%" notification="true" />
    <MonitoredMetric name="CPUBurst-Met-USCMS"
        method="http://URL/CPU?vo=USCMS&t=5"
        interval="5" type="%" notification="true" />
    <MonitoredMetric name="CPUBurst-Met-iVDGL"
        method="http://URL/CPU?vo=iVDGL&t=5"
        interval="5" type="%" notification="true" />
</Monitored>

    <!-- USTALAS minimal allocation -->
<Guarantee name="CPUBurst-G-USATLAS">
    <precondition usage="required">
        <consumer name="USATLAS-Prod" />
        <provider name="UChicago" />
    </precondition>
```

```
    <goal usage="required">
        <Constraint type="GreaterEqual">
           <Metric value="CPUBurst-Met-USATLAS" />
           <Value value="40" />
        </Constraint>
    </goal>
</Guarantee>

   <!-- USCMS minimal allocation -->
   <Guarantee name="CPUBurst-G-USCMS">
    <precondition usage="required">
        <consumer name="USCMS-Prod" />
        <provider name="UChicago" />
    </precondition>
    <goal usage="required">
        <Constraint type="GreaterEqual">
           <Metric value="CPUBurst-Met-USCMS" />
           <Value value="40" />
        </Constraint>
    </goal>
 </Guarantee>
```

```
      <!-- iVDGL minimal allocation -->
      <Guarantee name="CPUBurst-G-iVDGL">
       <precondition usage="required">
           <consumer name="iVDGL" />
           <provider name="UChicago" />
       </precondition>
       <goal usage="required">
           <Constraint type="GreaterEqual">
              <Metric value="CPUBurst-Met-iVDGL" />
              <Value value="20" />
           </Constraint>
       </goal>
    </Guarantee>
</uSLA>
```

Now, for the multi-market company example that could not be expressed completely in our first syntax (company level: *the computing nodes must be available to traders from 7:00 to 22:00, during the local trading day* and *70% of the aggregated resources from each individual market for Prediction computations whenever resources are not used by local users*; center level: *24 processors with at least 512 MB of memory and at least 2 GHz CPU-speed, interconnected with a 10 Gb/s switch are provided from 22:00*

to 7:00) is represented as follows (note in this example that time is collected as time + 2:00 and the comparison is then a simple *Less*):

```
   <uSLA name="Multi-Market Company uSLA (Scenario 2)">

      <!-- Define Monitored Metrics (and acquisition
mechanism) -->
      <Monitored>
      <!-- Time tracking -->
      <MonitoredMetric name="Time"
method="exec:/bin/date+2"
           interval="5" type="#" notification="true" />
      <!-- Company-wide monitoring -->
      <MonitoredMetric name="CPU-Prediction" interval="5"
           method="http://URLCompany/CPU?t=5&App-
Prediction"
           type="%" notification="true" />
      <MonitoredMetric name="CPU" interval="5"
           method="http://URLCompany/CPU?t=5"
           type="%" notification="true" />
      <!-- Center-wide monitoring -->
      <MonitoredMetric name="CPUType" interval="5"
```

```
        method="http://URLCompany/CPUType?t=5"
        type="%" notification="true" />
    <MonitoredMetric name="MemorySize" interval="5"
        method="http://URLCompany/MemorySize?t=5"
        type="%" notification="true" />
    <MonitoredMetric name="Net" interval="5"
        method="http://URLCompany/Network?t=5"
        type="%" notification="true" />
</Monitored>

   <!-- Company-wide allocations -->
     <!-- Any-application allocation rule -->
<Guarantee name="CPUBurst-CW">
      <precondition usage="required">
        <consumer name="ANY" />
        <provider name="Company-Name" />
    </precondition>
    <goal usage="required">
        <Constraint type="Less">
           <Metric value="Time" />
           <Value value="9:00" />
        </Constraint>
```

```
            <Constraint type="Less">
             <Metric value="CPU" />
             <Value value="100" />
          </Constraint>
   </goal>
</Guarantee>
      <!-- Prediction-application allocation rule -->
<Guarantee name="CPUBurst-CW-Prediction">
   <precondition usage="required">
       <consumer name="Prediction-App" />
       <provider name="Company-Name" />
   </precondition>
   <goal usage="required">
       <Constraint type="Less">
          <Metric value="Time" />
          <Value value="9:00" />
       </Constraint>
          <Constraint type="Less">
           <Metric value="CPU-Prediction" />
           <Value value="70" />
       </Constraint>
```

```
   </goal>
</Guarantee>

   <!-- Center-wide allocations (24 machines) -->
<Guarantee name="24 Machines">
   <precondition usage="required">
       <consumer name="ANY" />
       <provider name="Center-Name" />
   </precondition>
   <goal usage="required">
       <Constraint type="Less">
          <Metric value="Time" />
          <Value value="9:00" />
       </Constraint>
         <Constraint type="Equal">
          <Metric value="CPUType" />
          <Value value="24*2.0" />
       </Constraint>
       <Constraint type="Equal">
          <Metric value="MemorySize" />
          <Value value="24*512" />
       </Constraint>
```

```
        <Constraint type="Equal">
           <Metric value="Net" />
           <Value value="24*10" />
        </Constraint>
     </goal>
  </Guarantee>
 </uSLA>
```

Now, I turn my attention to show how the one example from the service outsourcing scenario is represented using the WS-Agreement like syntax:

```
 <uSLA name=”Service Outsourcing uSLA (Scenario 3)”>

    <!—Monitored Metrics (and acquisition mechanism) -->
    <Monitored>
    <!-- Time tracking -->
    <MonitoredMetric name="Time" method="exec:/bin/date"
         interval="5" type="#" notification="true" />
    <!-- Service monitoring -->
    <MonitoredMetric name="A-Requests"
         method="http://URL/A?metric=requests&t=5"
         interval="5" type="%" notification="true" />
    </Monitored>
```

```
   <!-- USTALAS minimal allocation -->
<Guarantee name="CPUBurst-G-USATLAS">
   <precondition usage="required">
       <consumer name="Grid3-Monitoring" />
       <provider name="Company-Name" />
   </precondition>
   <goal usage="required">
       <Constraint type="GreaterEqual">
          <Metric value="Time" />
          <Value value="7:00" />
       </Constraint>
       <Constraint type="LessEqual">
          <Metric value="Time" />
          <Value value="16:00" />
       </Constraint>
       <Constraint type="LessEqual">
          <Metric value="A-Requests" />
          <Value value="1000" />
       </Constraint>
   </goal>
```

```
    </Guarantee>
</uSLA>
```

3.5. uSLA Derivation from Local Site Usage Policies

While in the above section we provide a precise description for the uSLA syntax/semantics, usually usage policies at the provider/site level are expressed in terms of local RM configurations. Such usage policies are specified by resource policy makers; these uSLAs must be automatically discovered and translated for automated consumption at the other levels. To achieve this goal, I start and add methods and tools for automated translation to the high-level model for uSLAs described earlier in this chapter.

At the RM level (sites), *owners* state how their resources must be allocated and used by different *consumers*. This statement represents the *high-level GOAL an owner (MP) wants to achieve*. Site administrators map MPs to different software RMs' semantics/syntaxes. The end product is a set of RM configuration files, named *the local POLICY* or *system configuration (SC).* For automated consumption, site policies are translated from SCs by automated tools into an *abstract usage policy (AP)* set, i.e., the uSLA syntax/semantic described above. SC descriptions are collected from the site RM configurations, filtered and, after translation, published through a specific monitoring system, e.g., VO-Ganglia [67], MonALISA [13, 65] or GRUBER-SiteMonitor [43, 87].

I have identified three levels of description for the statement "*site X gives ATLAS 30% over a month*":

- MP: a description of a site manager's policy for the site, e.g., MP (VOs) = "*give ATLAS 30% over a month.*" I assume that simple English statements describe the MP set (site level);
- SC: an RM configuration: SC (VO) = <number of nodes, scheduler-type, scheduler-config>, which is written by the site administrators during the RM configuration process (site level);
- AP: An SC (VO) translated into a uSLA representation: *AP (VO, Site)* expresses *SC (VO)* in a scheduler-independent format and is published through a monitoring tool for resource brokering or other automated tools (Grid level).

The key point is determining how an SC maps to an AP. I have achieved this part by providing specialized SC-translators for each type of SC supported by a specific RM. The translator parses the configuration files or queries the resource provider RM, and outputs the resulting configuration directly into AP form. The information flow for this process is captured in a graphical way in Figure 3.8.

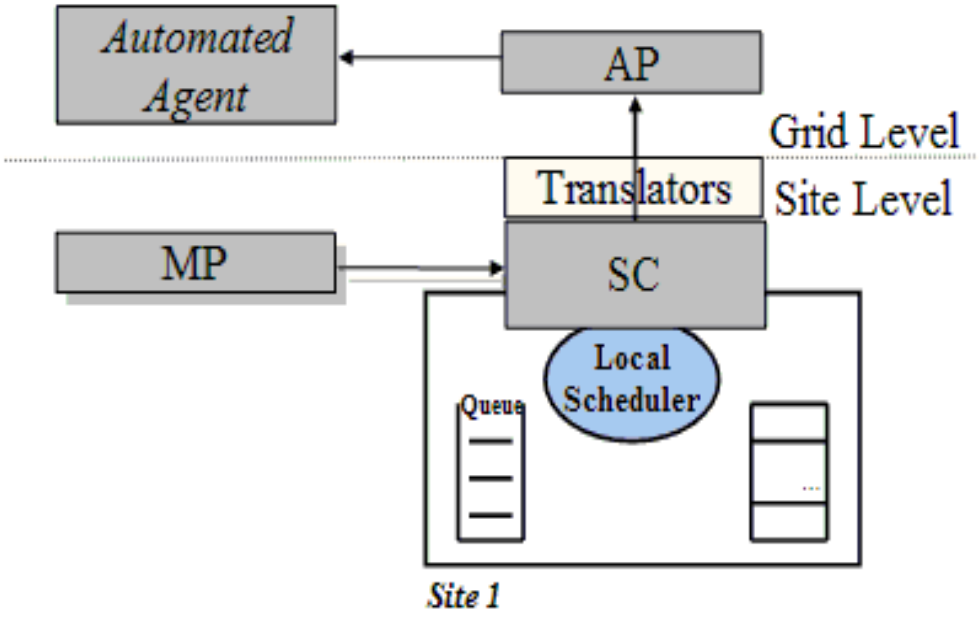


Figure 3.8: Correlations MP, SC, and AP

3.5.1. uSLAs Translation from Site Resource Manager Configurations

I provide now a concrete uSLA example, expressed in all the three forms, namely MP, SC and AP. I assume the following MP set for a site X:

- we have a cluster with 380 CPUs;
- at any time: USCMS-Prod has 40%; USATLAS-Prod has 40%; IVDGL has just 20% of these resources;
- when additional resources are available, Grid3's VOs can grab these resources;

The Condor priorities (SC) used to realize the above description is:

- `condor_userprio -setfactor cms 3`
- `condor_userprio -setfactor atlas 3`

- `condor_userprio -setfactor ivdgl 8`

In these settings, the AP description becomes:

- `RM type: Condor`
- `RM allocations: CMS: 40% ATLAS: 40% IVDGL: 20%`
- `Usage SLAs type: extensible-uSLA`

3.5.2. Disk Space Extensions

For disk space, the same allocations would be translated into the following rules (UNIX quota):

- `edquota -u cms`
- `edquota -u atlas`
- `edquota -u ivdgl`

The AP for disk becomes:

- `DM type: quota`
- `DM allocations: CMS: 40% ATLAS: 40% iVDGL: 20%`
- `UP type: extensible-uSLA`

This simple specification categorizes the storage into two types as a function of their data duration permanent storage (the hard limit quota limit), and volatile storage (the soft limit quota limit), as also introduced within the SRM project [95].

3.5.3. High Level Service Considerations

Next, I move to other types of services – more precisely Grid services. In this case, the service manager is either ARESRAN (a GT4 based prototype for complex resource

uSLA and reservation in Grids [83]) or any other similar management infrastructure [96, 97].

In this case, the uSLAs are collected directly from the authorization service and published at the Grid level. For the ARESRAN case, the uSLAs are already in a form that is compatible with the approach proposed in this dissertation, while for the other examples (SAML or the PDP policy service) they must be translated to a Grid level definition. I focus on the ARESRAN service, because they target a simpler kind of authorization mechanism – similar to the CAS system [14], mainly a flexible access control mechanism.

3.6. Summary

In summary, I have detailed in this section the scenarios in which uSLAs are required, the goals and requirements for these uSLAs, as well as supporting mechanisms (monitoring, syntax and semantics) to implement them in large and distributed environments. The results presented in this section can be grouped into three main categories: the identification of scenarios, the requirement for monitoring and uSLA mechanisms, and the uSLA proposed syntax and semantics.

CHAPTER FOUR

GANGSIM: A SIMULATOR FOR GRID SCHEDULING STUDIES

GangSim is a tool developed for Grid scheduling studies, capable of supporting studies for controlled resource sharing based on scheduling policies and uSLAs. I present, in this chapter, its design and simulation environment, and the kinds of studies it permits.

The starting point for this work was an exploration of distributed system monitoring conducted within the context of the GriPhyN/iVDGL projects [13, 98]. I started by developing the VO-Ganglia Monitoring Toolkit to gather resource characteristics, utilization data, and usage limits for a collection of sites in order to meet the monitoring requirements for uSLA-based resource provisioning described in Chapter 3. The final result was an enhancement of the Ganglia Monitoring Toolkit [88], the VO-Ganglia monitoring toolkit for large Grid environments. The Ganglia Monitoring Toolkit is a cluster monitoring tool that uses a multi-cast approach for monitoring various characteristics of a cluster and the availability of each individual host [88].

From Ganglia and VO-Ganglia, it was relatively easy to replace “real sites” with “simulated sites,” and thus to enable the evaluation of a wider range of possibilities for Grid scheduling than it is possible in a real system. The new name, GangSim [42], reflects both the origins of the implementation and the fact that it can be used to simulate “gangs” of users and resources.

GangSim’s novelty comes from its modeling not only of sites but also of VO users and schedulers, and its ability to model uSLAs at both the site and VO levels. The resulting simulation system allows various task assignment policies to be tested in conjunction with different uSLAs for a range of different Grid configurations and workloads.

This chapter is structured as follows. First, I present a short history about GangSim’s conception and describe the simulated environment. A scenario example is provided for better understanding of GangSim’s approach. Second, the implementation details and GangSim’s output types are presented. I end the chapter with a short analysis of its performance and the future work required to improve its results.

4.1. Simulator Model

GangSim simulates a uSLA-driven management infrastructure in which uSLAs for the allocation of resources within communities (VOs) and the allocation of resources across VOs at individual sites interact to determine the ultimate allocation of individual computing resources (CPU, disk, and network). uSLAs are expressed using the tuple

syntax introduced in Chapter 3 for rules associated with sites, VOs, groups, and users for different aggregations of available resources.

GangSim models a Grid as comprising a collection of sites, VOs and users, a job submission infrastructure, data files, a monitoring infrastructure, scheduling infrastructure, and an uSLA management infrastructure. The principal GangSim components are external schedulers (ES), local schedulers (LS), data schedulers (DS [75]), monitoring distribution points (MDP [33]), site policy enforcement points (S-PEP [33]), and VO policy enforcement points (V-PEP [33]). I describe each of these components in more detail in the next subsection.

4.1.1. Simulated Components

A *site* is characterized by the capacity and number of its CPUs, disk resources, and network capacities. Each characteristic is described through a configuration file that is loaded during startup. Both intra- and inter-site network capacities are defined. A site specifies a set of usage policies, defined by the uSLAs imposed by the site owner, for how much CPU time, disk space, and network bandwidth each VO may use.

A *VO* is composed of a set of groups of users. Users submit jobs, which may be grouped into sets called workloads. A VO specifies an uSLA that defines the resources that will be made available to each group.

External Schedulers, Data Scheduler, and *Local Schedulers* (ES, DS, and LS) represent points where various scheduling decisions are performed. An ES, such as Pegasus [99] and Euryale [100], queues user jobs and selects the best site candidate for each job. An LS, such as Condor [101], PBS [102] and LSF [103], queues jobs

associated with a site and schedules them on a suitable node for execution. A DS, such as Kangaroo [64, 102, 104, 105], schedules the required files to the candidate execution site. The ES interacts with LS in order to submit jobs to a specific site. A DS transfer files transparently to ESs and LSs, such that all required files for a job are in place when a job is ready for execution.

Monitoring Data Points (MDPs) represent the monitoring infrastructure "nodes" that compute various metrics for Grid components' consumption. Such information is gathered from local and external schedulers, filtered, and delivered in a uniform manner.

Policy enforcement points (PEPs) are responsible for enforcing uSLAs. They gather monitoring metrics and other information relevant to their operations, and then use this information to steer resource allocations as specified by the uSLA [26]. I distinguish two PEP types, S-PEPs and V-PEPs.

Site policy enforcement points (S-PEPs) reside at all sites and enforce site-specific uSLAs. S-PEPs operate in a continuous manner, in the sense that jobs are immediately preempted or removed when uSLA requirements are no longer met. However, jobs are not necessarily restricted from entering site queues just because an uSLA would prevent them from running.

VO policy enforcement points (V-PEPs) interact with S-PEPs and schedulers to enforce VO-level SLA specifications. They make decisions on a per-job basis to enforce VO uSLAs regarding resource allocations to VO groups or types of work executed by the VO. V-PEPs are invoked when VO schedulers make job scheduling

decisions to select which jobs to run, when to send them to a site scheduler, and where to run them.

4.1.2. Scenario Example

This section describes a scenario for how GangSim simulates job execution. In this scenario, there are three sites A, B, and C and one VO, V. Site A has no allocations for VO V, site B has a fixed-limit uSLA that provides 30% of its resources to V, and C has an extensible-limit uSLA that provides 40% of its resources to V. Let's assume that the current utilizations of the two sites providing resources to V at time t are as follows: site B executes jobs from VO V that sum up to 25% of the site CPUs and 10% of the disk space is used, and all jobs running at B sum up to 60% and 20% of the disk space; site C executes jobs from VO V that sum up to 35% of the site CPUs and 30% of the disk space, and all jobs running at C sum up to 70% and 80% of the disk space. In this environment, we consider the submission of three jobs with the requirements (expressed in percentages) as described in Table 4.1. I also assume that all the three input files are stored at site A initially (the percentages are expressed in the total resource capacities).

Table 4.1. Job Requirements (the numbers represent required number of CPUs per job and its utilization per site in percentages for the second column, and required number of files per job and space requirements in percentages for the third column)

Job Number	CPUs [#, %]		Disk Space [#, %]	
	Site B	Site C	Site B	Site C
1	1, 5	1, 8	1, 5	1, 6
2	1, 4	1, 7	1, 3	1, 5
3	1, 3	1, 4	1, 1	1, 1

The first job for VO V submitted to this environment at time t would be scheduled to either site B or site C. Let's assume that a first-fit scheduling policy is in place (the first matching site is selected for the next job execution). Under this policy, the new job would be sent to site B, and its utilization would go to 30% for CPUs and 15% for disk space for VO V. The required file is transferred by the DS from site A to site B before site B starts the execution.

The second job from VO V submitted at time t + dt (where dt is 1 second, for example) can be scheduled only on site C, because it is the only site still available for VO V, and its utilization goes to 42% for CPUs and 35% for disk. The required input file is also transferred from site A before the job starts its execution.

The last job for VO V submitted at time t + 2*dt can be scheduled only on site C (the extensible-limit uSLA allows more jobs as long as free resources are available; I assume that dt is small enough to ensure that jobs 1 and 2 are still running).

4.1.3. Implemented Strategies

During each simulation step, various algorithms and strategies are used to update the state of different components of the framework. There are algorithms for job selection, job assignment, and data file replication. There are also algorithms for computing the costs associated with different operations and for steering job flows through the framework. These algorithms are grouped in a single implementation module, and invoked whenever a decision regarding the state of a component needs to be made. Next, I describe GangSim's approach to handling the main elements of a simulated Grid environment.

Job flow: Jobs are submitted by users to ES queues according to its specific uSLA. Following site selection, jobs are moved to LS queues. A job may be rejected at the LS (e.g., because of lack of available disk space or the local policy for the submitting VO), in which case the job is returned to the ES queue to re-enter the scheduling process. If the job requires larger files that the capacity of any of the sites, it is rejected from the ES queue.

Costs: The simulator associates different time costs with each successful or failed operation. Thus, a job that starts after two rejections will incur the costs of two rejections and one successful submission, plus the time for input (no more than two files in our current implementation) and executable files' movement among sites and VO schedulers. The two input files are placed initially on a site node at random while the executable file is placed on the scheduler node when a new job is scheduled. Overall, the following time intervals are counted during a job submission:

- time to enter the scheduler queue (one simulator step);

- time for site assignment: ES queue computations (one simulator step plus time to wait for an available site);

- time for node assignment: LS queue computations (one simulator step plus time to wait for an available node);

- transfer time for job input files to the execution node: network allocation and transfer costs for the input files (if a job has an image of size S that is larger than the available bandwidth B for one simulator step, **(n+1)*B > S > n*B**, then the job will be transferred in **(n+1)** simulator steps);

- time for job transfer (job file image) to the node: network allocation and transfer for the executable.

For example, for the scenario introduced in the previous subsection, the first job will require one simulator step to enter the scheduler queue for VO V. Once the job is in the scheduler queue, it will be scheduled to site B in one simulator step. The required input files and job image are transferred (let's assume) in two simulator steps. Once this step is finished, the job waits at cluster B for local scheduling on an available node. The scheduling operation incurs another simulator step and the local transfer (let's assume) requires only one simulator step, because of better network connectivity. Now, the job is ready for execution. The total amount of time required for its startup was 6 simulator steps. The same steps are repeated for the second and third jobs, but these steps overlap for the three jobs because they are submitted within a short time of each other (dt smaller than half of a simulator step).

Simulator steps: GangSim is a discrete simulator, which means that every **X** seconds the simulator evaluates the state of all components in the system (jobs, queues, resource status, allocations, utilizations, etc). Operations take place only during evaluation steps. If a job has a running time **(n+1)*X** > t > **n*X**, then the job occupies a resource for **(n+1)*X** seconds, but the utilization accounted against the job owner will still be **t** seconds (by accounting only the job's required execution time). The consequence is the smaller the value of **X**, the higher the accuracy of the simulator. I have typically used a value of 5 to 10 seconds in my experiments, except the scenarios of Sections 4.8 as detailed later. Such a value for parameter **X** appears to be acceptable as long as the average running time for jobs is greater than or equal to a few hundred seconds (see section 4.3.5 for more analyses).

VO and Site uSLAs: uSLAs are expressed as tuples as was described in Chapter 3.

Task assignment strategies: The effectiveness of a specific uSLA may also depend on the strategies used by ESs and LSs. Based on De Jongh's [106] scheduling taxonomy, we distinguish between *static* and *dynamic* scheduling policies. Static policies use only *a priori* information about a system and workload. For instance, such policies either ignore utilizations or consider them known before hand. On the ES level, I focused on dynamic scheduling policies, such as least-used, least-recently-used, that take into account site loads, and on static scheduling policies, such as round-robin, random-assignment [106]. The input to these policies is filtered by uSLAs to use only sites providing allocations. For the LS level, GangSim simulates a first-come-first-served and space-shared scheduling. Space-shared denotes a scheduling policy under

which only one job runs on a node until either completion or migration to another one. The best analogy for the current UNIX OS is the memory resource (the space allocated to a process can not provided to other processes).

4.2. Implementation Details

In this section I describe both GangSim's main modules and their interactions in order to perform a simulation scenario. I start with first the description of the main modules, followed by the description how they are invoked for execution.

4.2.1. Main Components and Their Functionalities

VO-Ganglia uses several reporters (UNIX tools that provide different system monitoring metrics) for information gathering. In GangSim these reporters are replaced with modules that simulate a Grid environment. Because the GangSim's reporters need a description of the simulated environment, I have developed additional tools for Grid environment generation and workload specification. The environment generators create configuration files that describe the number of sites and the configuration of these sites based on a high-level description of the Grid. The workload generators create configuration files describing, in detail, workloads based on the Grid configuration files and a description of job distributions. Site uSLAs can be either specified through a web interface in a centralized manner (once per simulator instance), or described in a configuration file with a higher level of granularity (one per site). The internal modules of GangSim are presented in Figure 4.1.

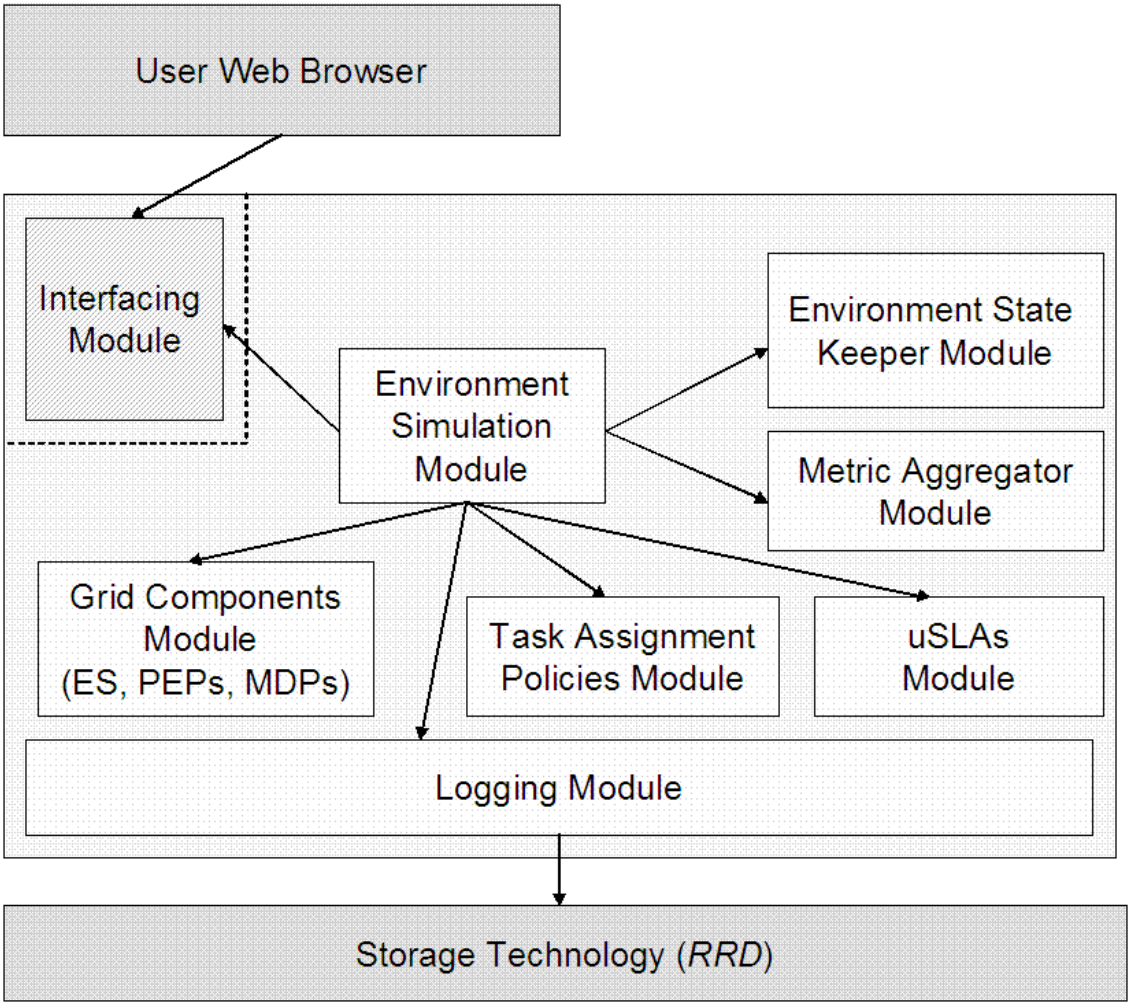


Figure 4.1: Simulator Overview

Each component has the following purpose. The *environment simulation module* is a set of Perl libraries that maintain the Grid environment status for workloads and generate appropriate metrics to feed GangSim. This module simulates ESs, submitting hosts, LSs, and sites. MDP data is provided to the module instead of (or, in a mix-mode execution, as well as) real ESs, LS, S-PEPs, and V-PEPs. *Task assignment policies* represent a set of algorithms that can be invoked for scheduling jobs to sites, scheduling jobs to nodes, or for selecting jobs to run. These algorithms are used by S-PEPs, LSs, V-PEPs, and ESs. Similarly, the *uSLAs module* provides a set of already identified uSLAs algorithms for environment controlled sharing.

The *metric aggregators* are a set of routines that aggregate metrics based on rules such as, string concatenation, integer median computation, or integer averaging. *Grid components* represent a set of functions and data structures for simulating Grid components. For example, queues are represented by arrays of structures for various metrics about jobs; sites are modeled as a list of physical node capabilities and instantaneous states; workloads are also maintained under various queue structures as they pass from one stage to another in the simulated environment.

Environment State Keeper maintains a set of data structures that hold data used for system simulation, including: workload status, Grid component status, and current utilization. The user *interfacing module* is composed of a set of CGI scripts that gather GangSim status information and present it in HTML form. The last module, *logging module*, represents the connection of the entire simulator with an underlying technology for logging simulated environment states. Currently, it allows only RRD-based [1, 88]

interfacing for saving historical environment data, and text files for tracking the operations performed by the simulator.

While simulating various size Grids under different uSLAs, scheduling policies, and workloads would be sufficient in most cases, I have also introduced the possibility of simulating different Grid system types. Specifically, I mean Grids that use different approaches in how the uSLAs are managed. At present, GangSim can model two types of Grid systems: uSLA *analytical-based scheduling* and uSLA *observational-based scheduling*.

In the first case, uSLAs from all participants are available to ESs. In this case, job scheduling decisions are based on direct knowledge of the uSLA.

In the second model, job scheduling is determined by observing the scheduling operations at each site. Because no knowledge about allocated resources is available at the ES level, each ES monitors the number of jobs that can be run by a site. Further, two variations of this model were pursued: *no-memory* and *memory-based*. For the no-memory approach, whenever new jobs can no longer be scheduled at a site, ESs stop submitting jobs to that site. For the memory-based variation, GangSim maintains the observed limits over the time. Whenever jobs from a given VO no longer start at a site, the burst limit is inferred. As soon as the number of jobs from a given VO running at a site drops to zero and no other job is started, the epoch limit is inferred as well.

4.2.2. Simulation Approach and its Limitations

From an execution point of view, GangSim uses two threads for achieving its goals. One thread, the *collector*, is used for serving external queries based on the state and

instantiates the *interfacing module*, while the other thread, the *executor*, is used for: (a) computing the next environment state, (b) updating by means of the logging module, the historical environment trace, and (c) providing the environment state to the collector thread. While the collector is reduced in technical complexity and implements an event driven mechanism (provides answers when external queries are performed or updates its environment view when signaled by the executor), the executor performs the three main operations described above for every simulated step in order to update the simulated environment state at a given time.

GangSim, like Ganglia, stores logged data in RRDs [1]. The RRDs are created with a pre-specified data update time interval for each simulation run, and in the current implementation GangSim, the RRD interface requires input data for every update interval [1, 88]. Because of this RRD logging implementation, to calculate accurate results GangSim *must* perform each environment state computation in a time interval, which I call the *required time*, lower than or equal to the simulated time step. In other words, if the executor requires more time than the time step, then under the current historical logging technology the simulator will produce inaccurate results. Given the following notation:

t – time step

s – time step size

r – required time

S – start time for time step t (S = s * t)

the following two scenarios are possible when simulating large Grid environments:

- Accurate: when the executor is not overloaded at time S (r <= s), the RRDs are updated and the computation for the next step (t + 1) will begin at time S + s; for the remaining time (s - r), the executor will sleep;
- Overload: when the executor is overloaded (r > s), the computation for the next step (t + 1) will begin at time S + r. The RRDs for the current step (t) will be updated at step t + 1.

Because of incomplete information in the RRDs due to overloading gaps are introduced in the graphical results, and the performance metrics have discrepancies. In the overloading scenarios, the simulated time (*Grid environment time*) differs from the real time (*GangSim environment time*).

4.3. GangSim Output

In this section, I explain the simulator features of GangSim using several examples. I assume that each site has a predefined number of CPUs, and each VO a predefined number of groups that submit workloads. I use synthetic workloads to validate the simulator, evaluate its performance, and provide examples of its capacities. Each workload is composed of jobs, each corresponding to a certain amount of work and with precedence constraints (job ordering in a workload) determining the order in which jobs can be executed [38, 106].

The output of GangSim is represented by three types of information: performance metrics, graphs illustrating simulation history, and instantaneous display environment pages (HTML and XML documents). Performance metrics are important because they

provide support for qualitative analysis of how workloads are executed in a specific scenario. The graphs represent a means for tracking the correctness of uSLA enforcement. Finally, the instantaneous display pages allow for combining the simulator with other tools (like real Grid schedulers) and quick user verification.

4.3.1. Experimental Setup

The scenario and experimental setup considered in the rest of this chapter is as follows. Three VOs each submit a workload to a single site. The VOs' workloads are composed of 28, 34 and 39 jobs, respectively, under a Poisson distribution; the length of the jobs is around 200s based on a Gaussian distribution (the length of jobs in a workload is distributed with the average equal to 200s); the input files have size between 1kb and 5kb. For all the simulations, each job requires as input 2 files from a set of 5 to 20 (a random number) files placed (randomly) on a site's node. Also, for the analysis of uSLA-based resource sharing, job precedence constraints do not have any impact on how resources are provided; these precendences will just postpone the execution of later jobs, while my focus is on how actually resources are shared. The site has 28 machines shared under one of the four uSLA semantics introduced in Chapter 3, with the simplification for the commitment-limit uSLA that a burst can last as much as the epoch limit is not reached.

The simulation interval is 10 minutes in all scenarios with a simulator step of 5 seconds. The job scheduling strategy for both the VO and the site levels is first-come-first-served (FCFS). The job assignment at the VO level is round robin while at the site level is round robin combined with the close-to-file policy. When at least one of the two

input files is already at a node, and the node is free, then this node is selected. The close-to-file scheduling policy influences workload executions by lowering the time for input file acquisitions. When a file is already at a site due to previous executed job dependencies, there is no need for a subsequent movement as long as the file is still at the node; it can be reused by subsequent jobs that require it. However, a file can be deleted from a node if the node runs out of space and it is not the initial source where the file was firstly created.

As a final note, all simulations are performed on an AMD Athlon™ XP 3000+ with 1 GB of memory, and the epoch allocation is considered the simulation interval (10 minutes).

4.3.2. Performance Metric Examples

Performance metrics provide support for quantitative analysis. GangSim supports the following six metrics: *aggregated resource utilization* (**Util**), *job completion* per site, VO or overall (**Comp**), *average site response time* (**Delay**), *average Grid response time* (**Response**), *average starvation factor* (**Starv**), and *uSLA violation ratio* (**Violation**).

Table 4.2 captures the six metric values captured for the four scenarios introduced in the previous sub-section. As can be observed, the no-limit uSLA offers the lowest **Response** under the FCFS scheduling policy employed. At the same time, it is the second best uSLA in terms of the total system utilization, but the difference between no-limit and commitment-limit uSLAs is minimal (2.1%). This difference is explained by the need for different input files in the no-limit case: more jobs from a single VO start

and thus more bandwidth is needed to transfer a file from the node where it resides to multiple destinations (nodes in the same cluster). The other two uSLAs have lower average resource utilizations and higher average response times for this scenario, because fewer jobs are scheduled due to the uSLA limits.

Table 4.2: Response and Util Value Results

Metric\uSLA	no-limit	fix-limit	extensible-limit	commit-limit
Response (s)	9.18	14.16	13.53	10.91
Util (%)	68.55	61.19	65.85	70.71
Comp (%)	80%	80%	80%	82%
Delay (s)	6.97	6.11	6.58	6.85
Starv (%)	8.49	12.32	10.38	7.82
Violation (%)	*17.15*[*]	0.0	6.90	12.01

[*] *With the assumption that resources are equally shared among the VOs (each should get a 33%of these resources)*

I introduce two additional metrics to measure GangSim's overloading. Simulated step requirements (**Required**) represents the ratio of required time to the time step size for completing the computations required by the Grid environment state update. In addition, *average required* is defined as the average of the required metric over an entire simulation interval. Whenever the required metric is smaller than or equal to one, the simulator is not overloaded.

System overloading metric (**Overload**) captures the level of inaccuracy introduced into the simulator by the limitations of the logging facility:

$$\textbf{Overload} = 1 - n * s / t$$

where **t** is the GangSim environment time, **s** is the time step size, and **n** is the number of steps successfully simulated at time **t**. A small overload time indicates that Grid environment time is close to the GangSim environment time; whereas a large value indicates that the Grid environment time is well behind GangSim's environment time.

4.3.3. Graph Output Examples

In this section, I provide an example of GangSim's graph output. For each uSLA, four figures are presented: the first three represent CPU utilizations (burst and epoch) for VO_0, VO_1, and VO_2, respectively. The fourth figure represents the aggregated CPU utilization (again, burst and epoch) for the three VOs. In addition, the uSLAs are represented for the last three scenarios as lines that express either burst or epoch allocations. The two axes are CPU utilization (expressed as percentages on the vertical axis) and simulated time (expressed as seconds on the horizontal axis). In addition, when a limit is enforced, it is also represented on the left axis as a percentage.

The first simulation represents the execution of the sample workloads on the site's resources for a no-limit uSLA. Each VO acquires CPU resources as it submits jobs into the schedulers' queues. Figure 4.2 shows that VO_2 acquires fewer resources in the beginning (at simulated time 120s), even though the various VOs' workloads have comparable resource requirements (20% for VO_2 compared with 42% for VO_0 and 37% for VO_1, thus there is over 100% imbalance between VO_0 and VO_2). The explanation is that in absence of any limitations, VO_2's submitted jobs are scheduled only after the other VO's jobs submitted earlier are finished, because the third VO starts submitting in jobs at a later time.

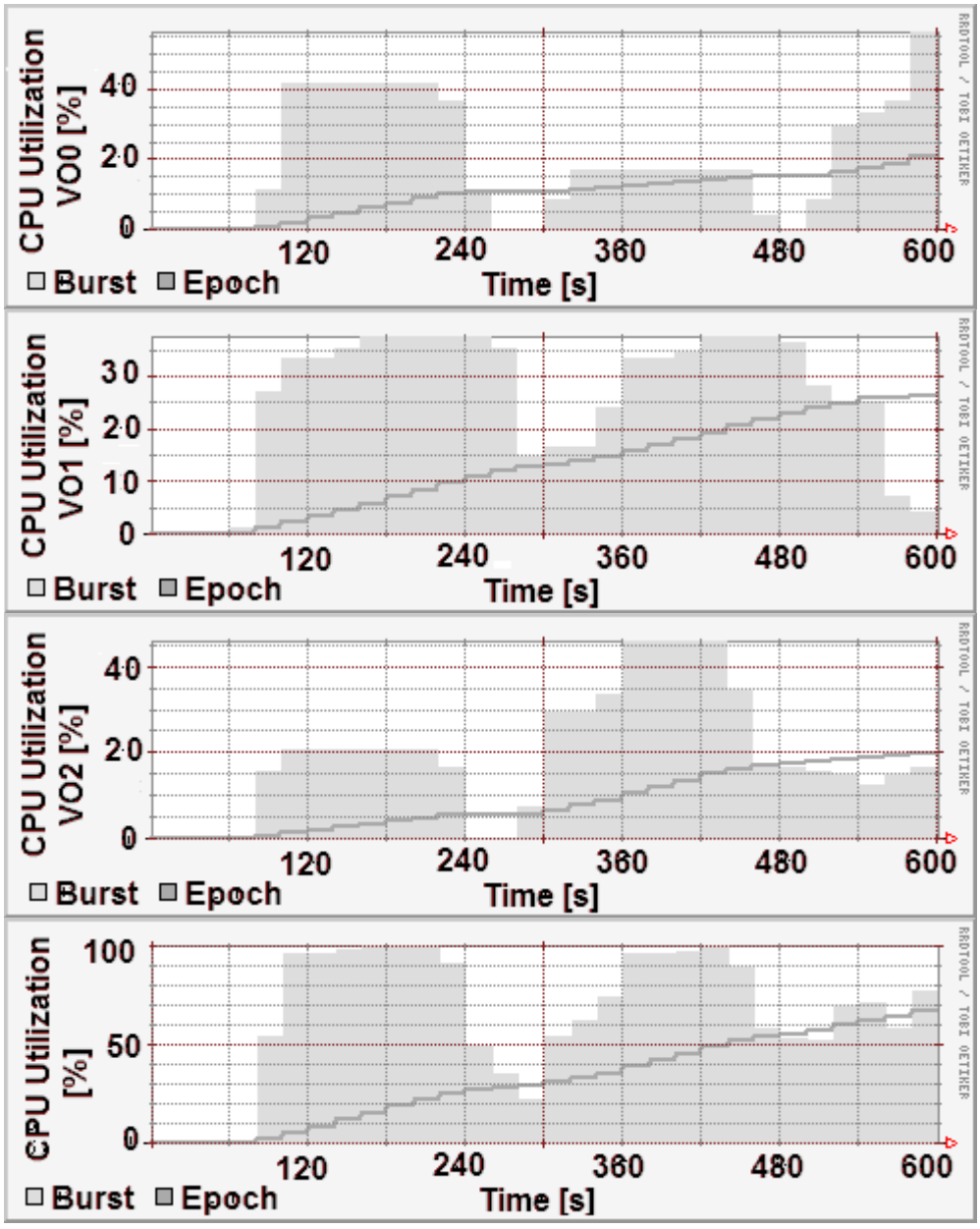


Figure 4.2: No-limit uSLA Simulation Example

The second simulation represents the execution of the sample workloads on the site's resources for a fixed-limit uSLA. The uSLA imposed in this scenario allows at most 30% for each VO and a total of at most 90% for all the VOs. In this scenario, each VO acquires at most its allocated share of the CPUs as can be observed from Figure 4.3. For this scenario, each VO acquires an equal share, 30%, of the total CPU resources. However, even though additional resources were available, they remained unused, because the fixed-limit uSLA allowed the three VOs to consume only 90% of the site resources.

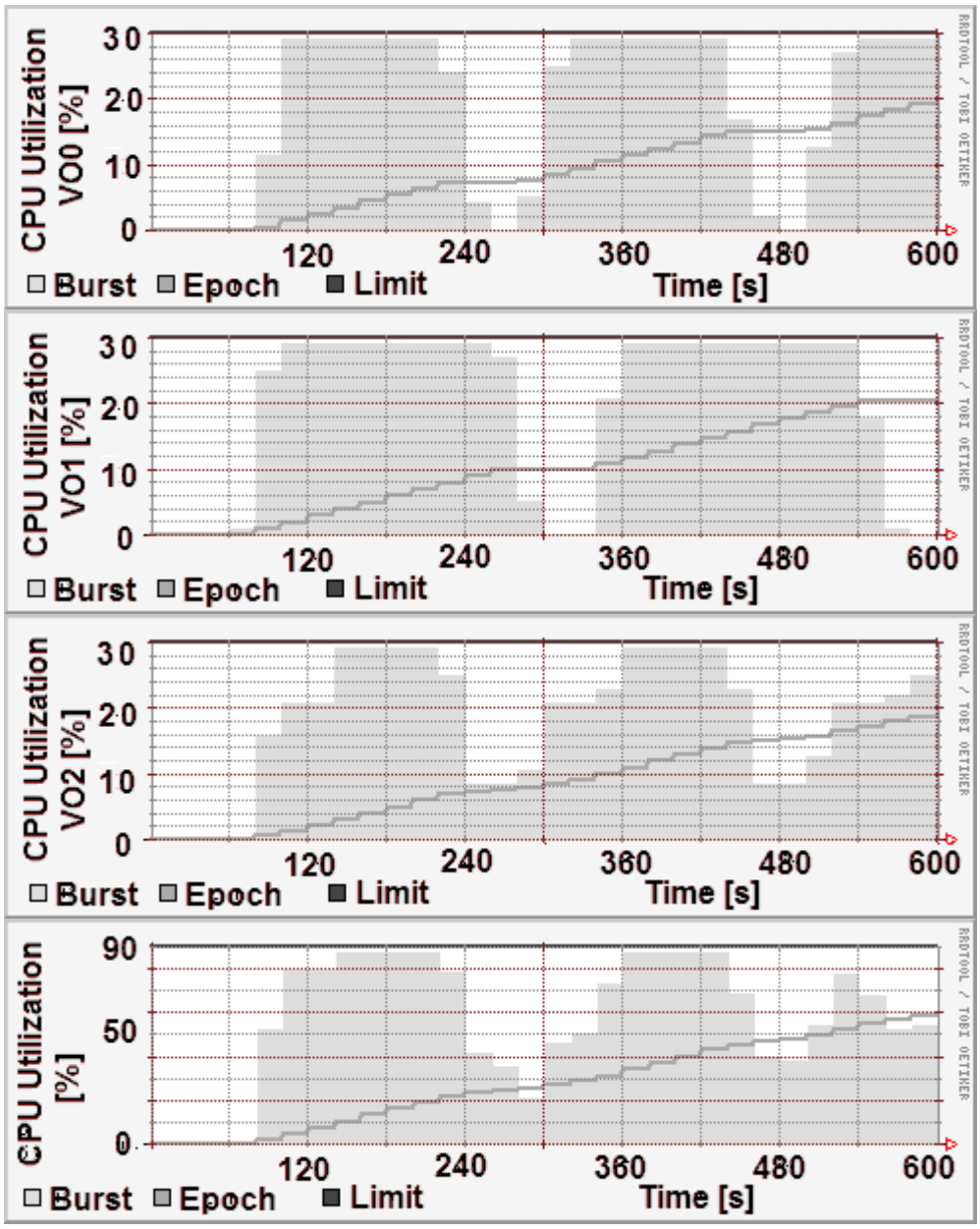


Figure 4.3: Fixed-limit uSLA Simulation Example

The third simulation represents the execution of the sample workloads using an extensible-limit uSLA. The uSLA imposed in this scenario allows each VO to use as many resources as possible when there is no contention, but at most 30% when contention occurs. In this scenario, each VO acquires at most its allocated share of CPUs over the epoch, as can be observed from Figure 4.4. The difference with the previous scenario is that while all VOs acquire at least their entitled share, the VOs that submit their jobs earlier get additional resources. Each of the three VOs acquire, at different intervals, more resources than their allocations: VO_0 has spikes up to 41%, VO_1 up to 37% and VO_2 up to 37%. However, the epoch CPU allocation is comparable for the three VOs (each VO acquires utilizations in the 20% - 25% range over the 10 minute interval).

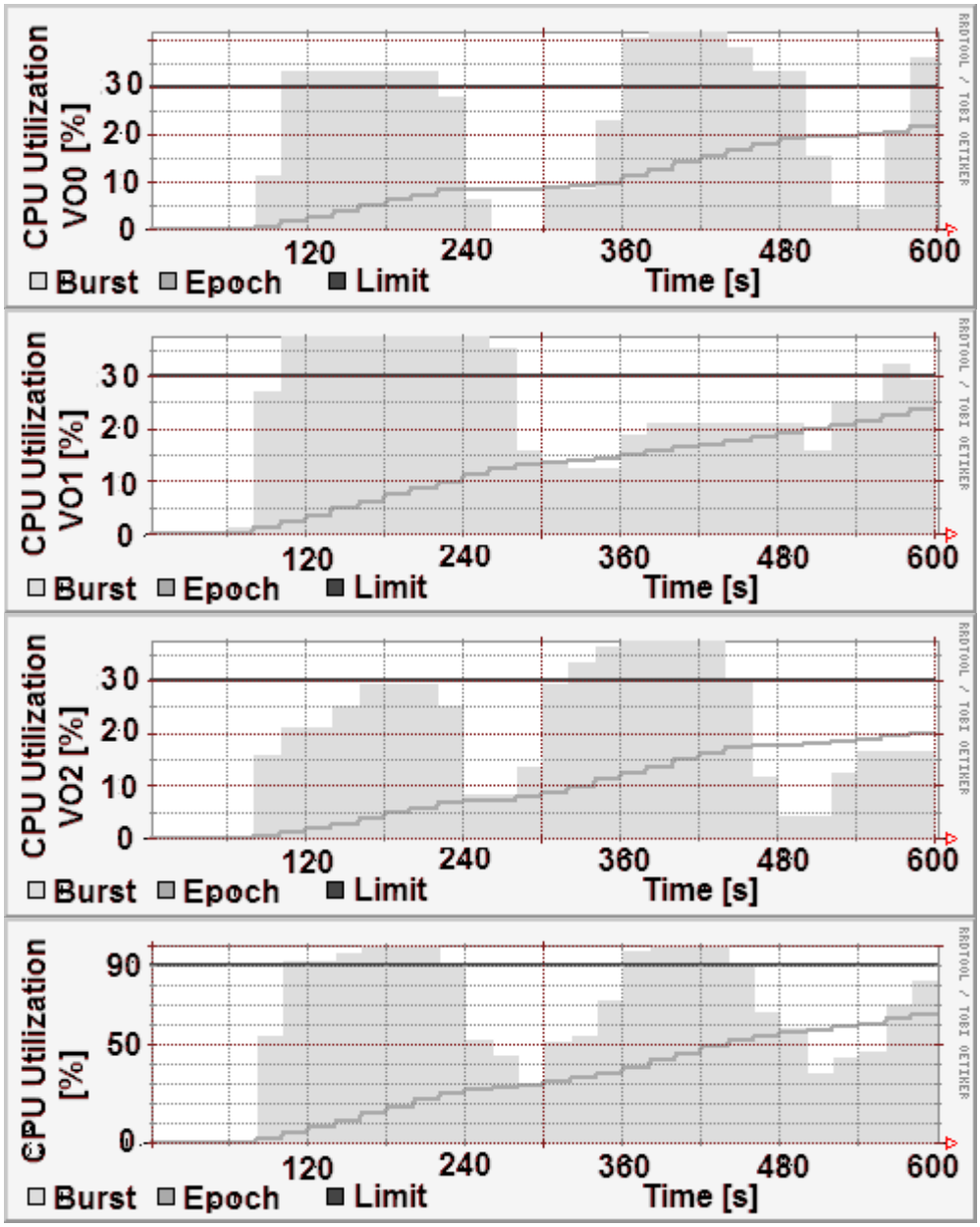


Figure 4.4: Extensible-limit uSLA Simulation Example

The final simulation represents the execution of the workloads using the commitment-limit uSLA. The uSLA imposed in this scenario allows VO_0 at most 30% for the entire period with spikes up to 60%, VO_1 at most 30% with spikes up to 60% and VO_2 at most 30% for the 10 minute simulation interval with spikes up to 50%. In all cases the spikes can last until the epoch allocation is consumed. In this scenario, the VO with the most jobs in the queue can have spikes in its instantaneous allocation, but will never exceed its burst limit.

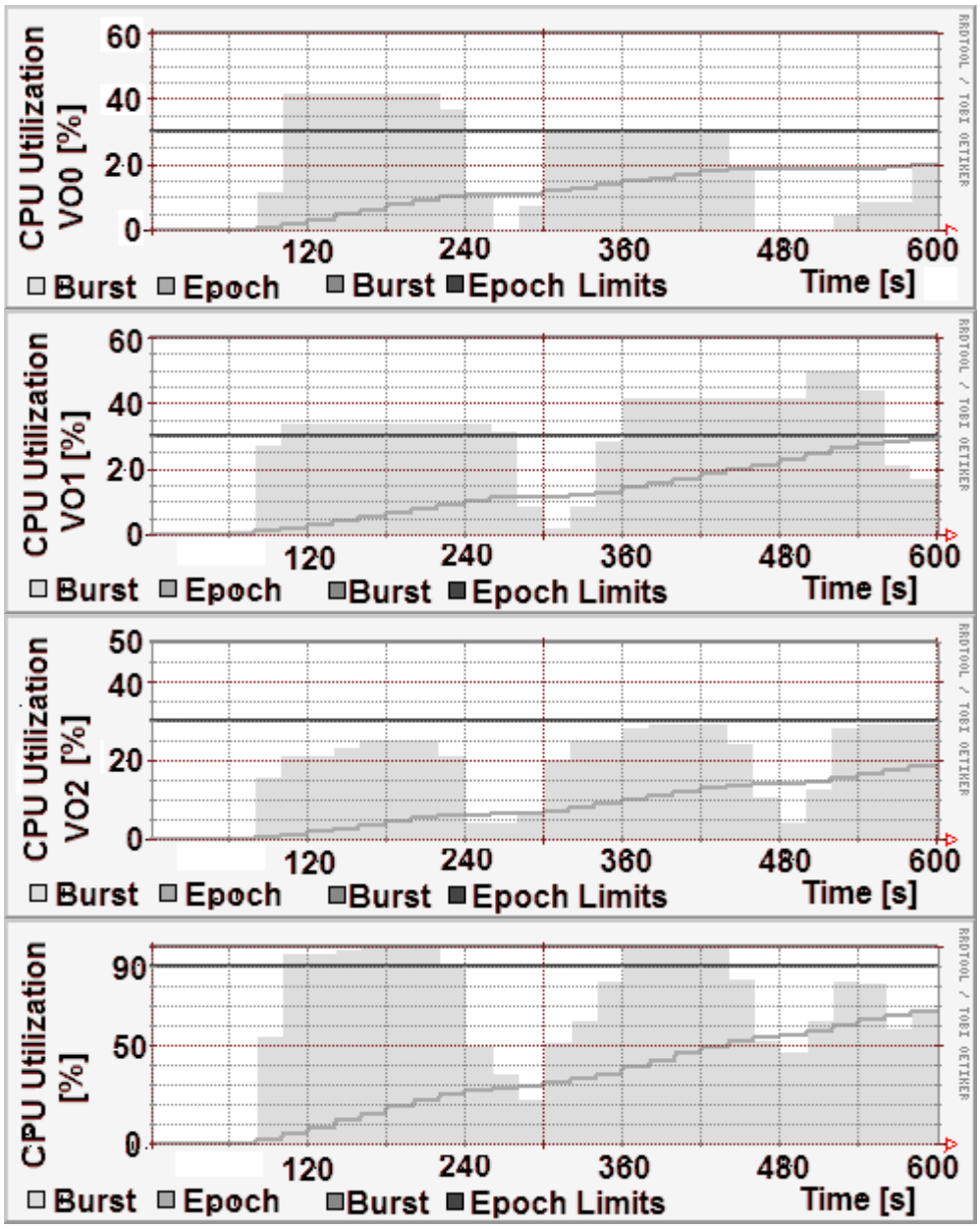


Figure 4.5: Commitment-limit uSLA Simulation Example

In conclusion, I have presented GangSim graphical output for the four uSLAs introduced in Chapter 3 and the scenario of section 4.3.1. From these results, the commitment-limit uSLA is the best sharing approach of the four methods, because it achieves the highest resource utilization. This conclusion is also supported by the values from Table 4.2, which shows that the VOs achieve the lowest starvation, second smallest response time, and the highest overall resource utilization, under this uSLA.

4.3.5. Instantaneous Results

Simulation results are accessible instantaneously through a web interface. This web interface offers a simple and easy way to browse and view statistics about various components in the simulated environment. There are three main screens, the site view, the VO uSLA view, and the scheduler level view. Each view has many associated sub-views for monitoring a particular component. For example, a user can inspect how a scheduler assigns jobs to a site.

Figure 4.6 shows an example of the external schedulers' interface for the fixed-limit scenario at simulated time 120s. For each VO scheduler, the scheduled and pending jobs are represented on the left column. The VOs and their jobs are represented on the second column. The right column represents the scheduled and waiting jobs for each VO's group. The numbers for the first line on the bars have the following meanings: 7 stands for the total jobs in execution scheduled by means of $Scheduler_{0\text{-}0}$, 19 for the total jobs scheduled by means of the same $Scheduler_{0\text{-}0}$, while 29.16 stands for the burst percentage of resources consumed by the jobs scheduled by means of the same

scheduler from the entire Grid. Further, the numbers have the same meaning per VO and VO group.

Figure 4.6: The 'Schedulers' View Exemplified

4.3.6. Simulation Step Value Influence

The last element of the analysis is the influence of the simulation step value on the performance of the simulator. We performed additional runs for the scenario described in Section 4.3.1 with two different values: 30s and 60s vs. the 5s setting used before. The new simulation results (graphs) are captured in Figure 4.7 and Figure 4.8. It is clear that the increase in the simulator step affects the delay in job scheduling within the simulated environment. The difference is quantifiable as three simulator steps, based on the GangSim cost descriptions provided in Section 4.1.3. For each scheduling operation, at least one simulator step is required, and because the steps are 6 and 12, respectively, times larger, the jobs startup is delayed accordingly.

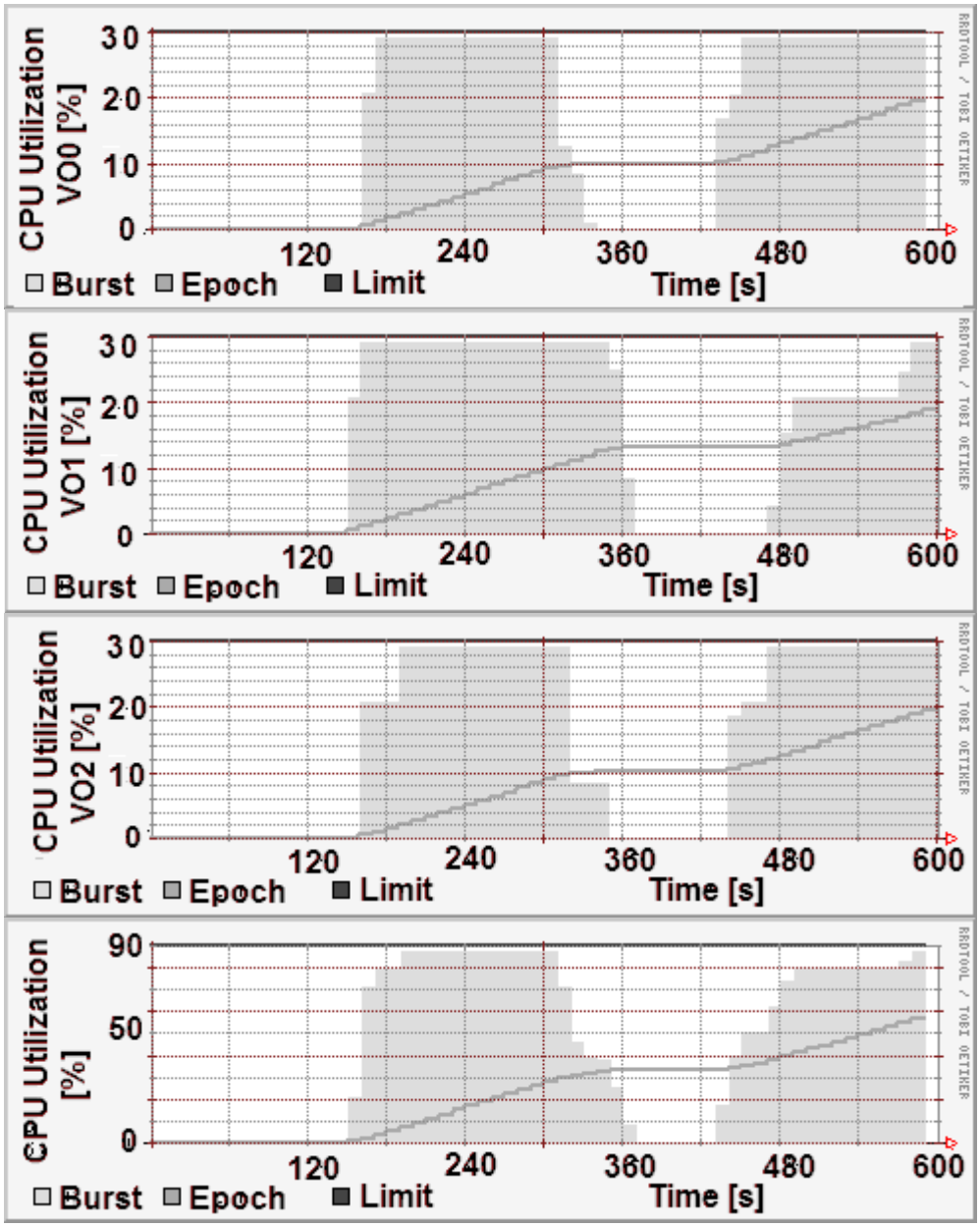


Figure 4.7: Simulation Results for a 30s Simulation Step with Fixed-limit uSLA

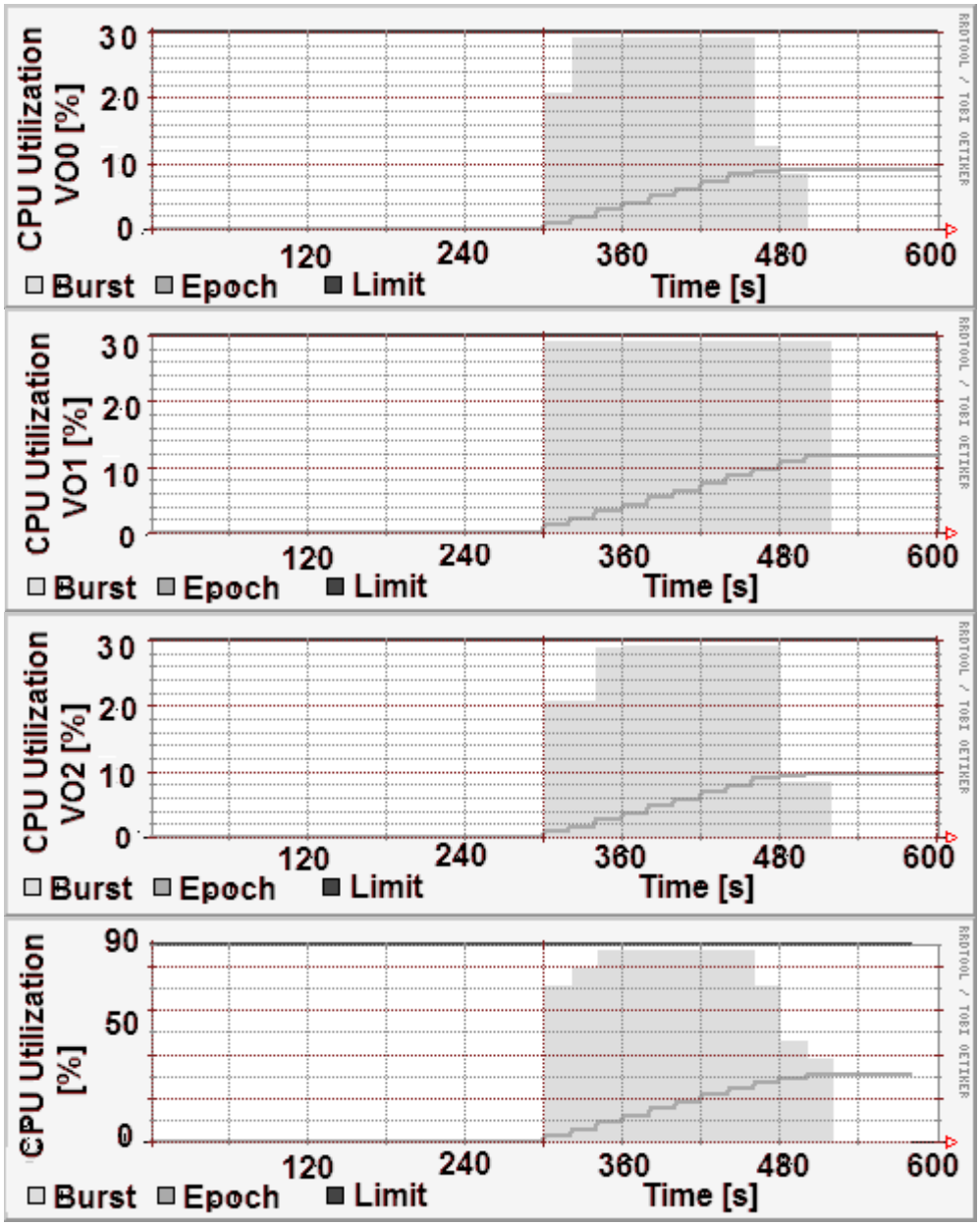


Figure 4.8: Simulation Results for a 60s Simulation Step with Fixed-limit uSLA

4.3.7. Simulated Architecture Variations

Figure 4.9 and Figure 4.10 represent two simulations of the scenario presented Section 4.3.3 based on *uSLA observational scheduling* for a fixed-limit uSLA, which is the easiest uSLA to compare from a reader's view-point. In both cases uSLAs are not available: in the first case, the ESs try to maintain at least one job waiting in the local site queue; in the second case, the ESs observe the number of jobs running concurrently for each site and each VO and infer the uSLA at the site. These extra jobs increase the queue load at the execution site without actually starting to execute. As a consequence, the other VOs (VO_2 and VO_0, in our scenario) schedule fewer jobs on the site and resources remain partially unused.

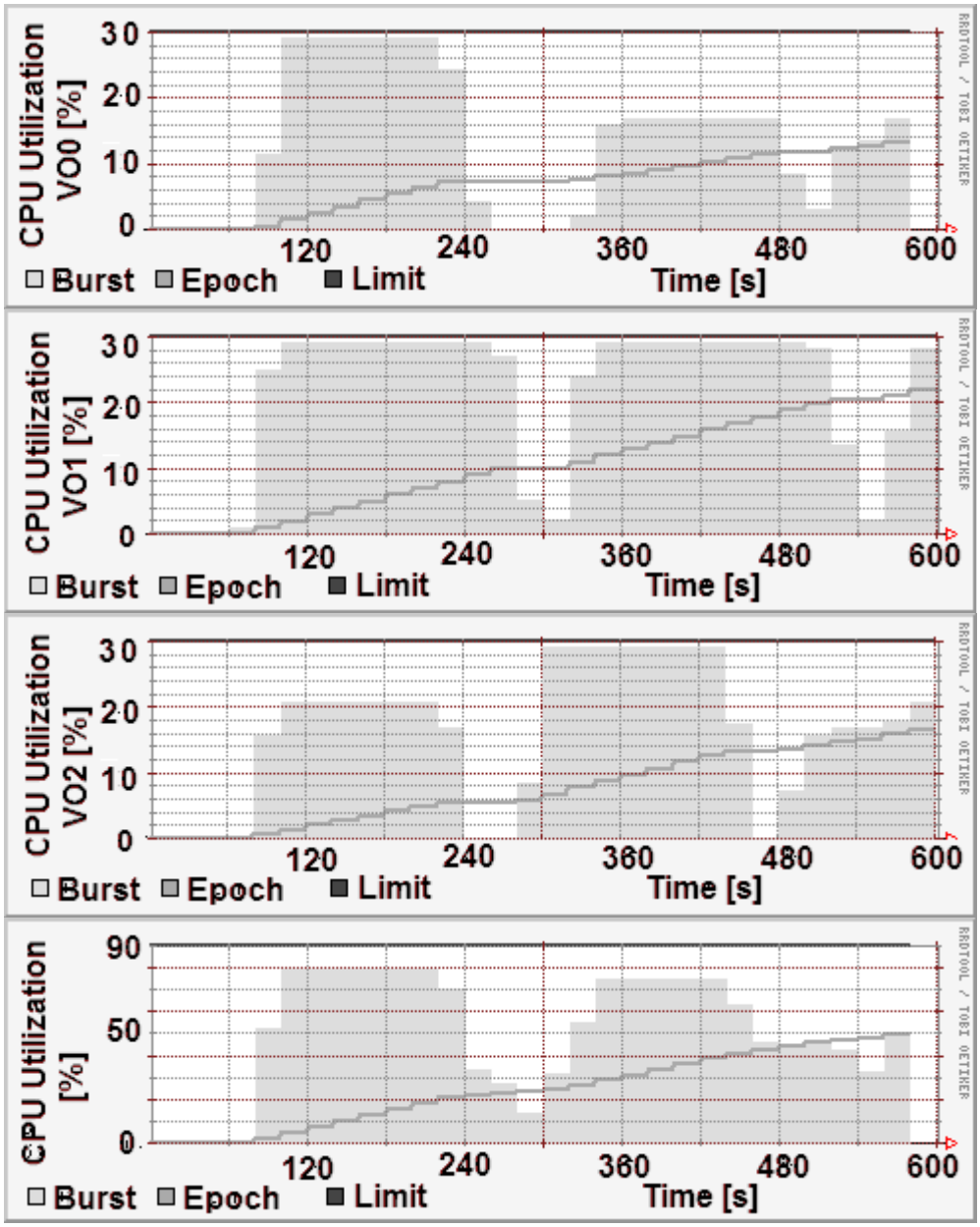


Figure 4.9: Observational Approach (no-memory) for uSLA Discovery

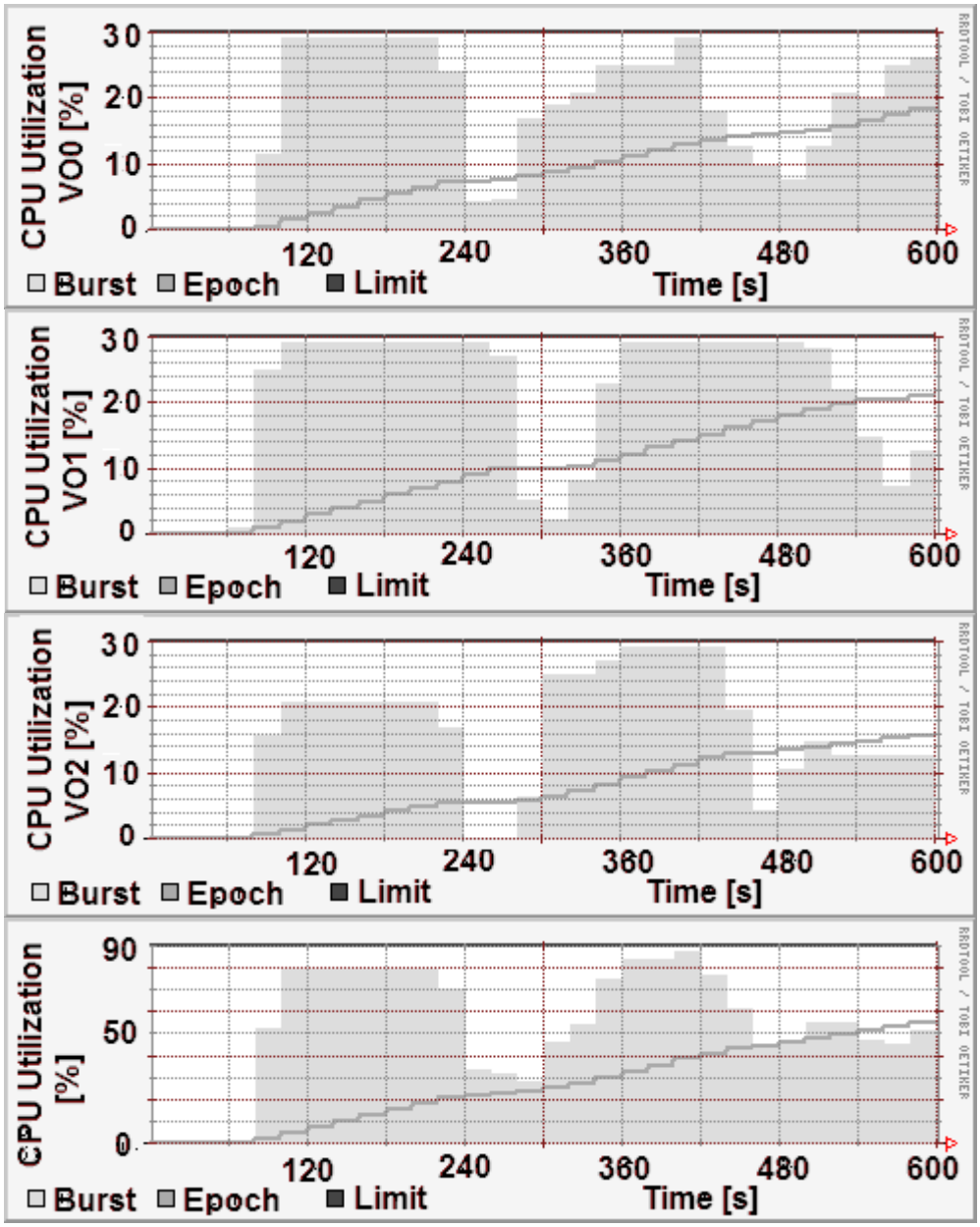


Figure 4.10: Observational Approach (with memory) for uSLA Discovery

While the two figures provide meaningful visual information for quantifying the behavior of the two observational approaches, I provide the **Response** and **Util** values for the three scenarios in Table 4.3 for comparisons purposes. As the results show, the analytical approach outperforms the other two approaches in our simulation scenario. While the simple observational approach cannot detect all the available resources based only on monitoring information, the memory-based approach comes closer to learning these limits (the consumption of the three VOs never gets to the point where it spikes up to the 90% limit).

Table 4.3: Response and Util Value Results (fix-limit uSLA scenario)

Metric\uSLA	analytical	observational (no-memory)	observational (memory)
Response (s)	14.16	18.21	17.02
Util (%)	61.19	49.92	55.21

4.3.8. Simulator Performance

In this subsection I show by experimentation that the required time for a simulation is dependent on three factors: number of participating sites, workflow sizes and number of participating VOs. I consider next three scenarios for evaluating GangSim scalability and performance, one for each of the above mentioned factors. In all three scenarios, the scheduling policy is random assignment, the uSLA is the extensible-limit uSLA (50% for each VO), while the simulation step is 5 seconds. A variable number of VOs

submit workloads composed of pre-defined number of jobs (200 per VO if not stated), with a constant arrival rate (2 jobs per second) for a 1200 seconds time interval.

To measure the impact of the environment size on the quality of results, six simulations were performed for 15 VOs with workloads composed of 200 jobs each, and for different environment sizes: 100, 125, 150, 175, 200, and 225 sites. Figure 4.11 captures the average required metric for each environment size. The first figure shows that in average GangSim still copes with the computation requirements for 100 sites, 15 VOs and 200 jobs size workloads per VO, but the required time increases substantially as the size of the environment grows.

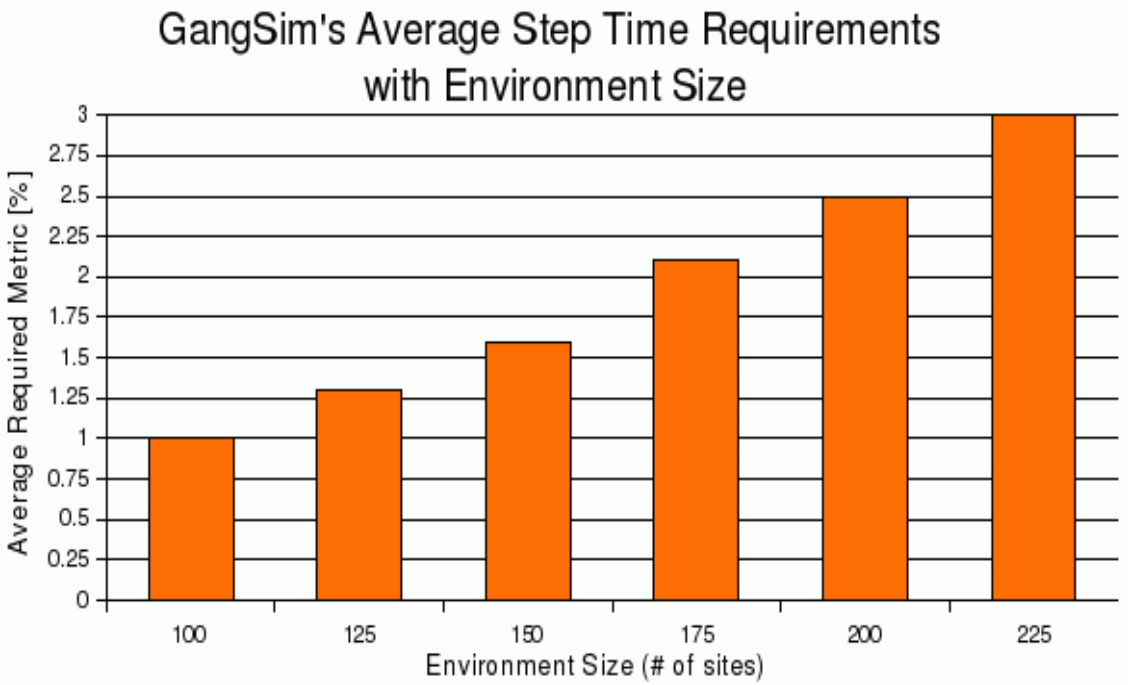


Figure 4.11: Average Requirement Metric Function of the Environment Size

Figure 4.12 presents the average overload metric as a function of the number of sites (100, 125, 150, 175, 200, and 225) at four GangSim environment times: 300s, 600s, 900s, and 1200s. While its value is close to 0 for 100 sites (low overload), it increases quickly as the environment size grows.

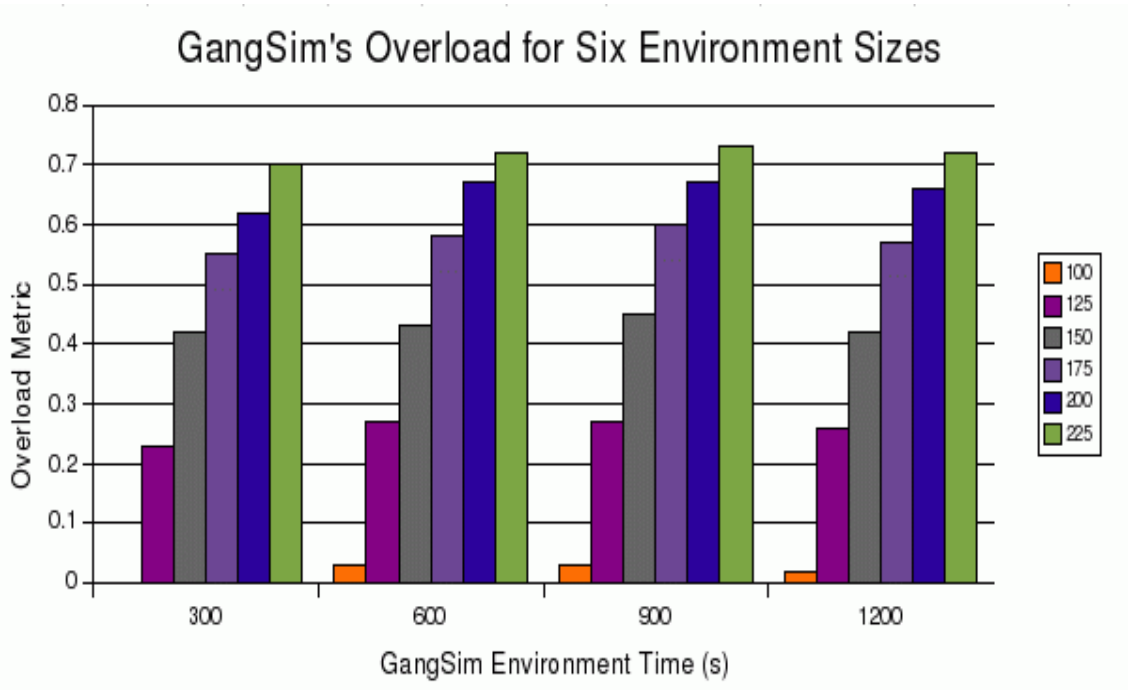


Figure 4.12: Overload Metric Function of the Number of Sites during Executions

To evaluate the impact of the workload size, four simulations were performed on a Grid of 225 sites and with 15 VOs for different workload sizes: 100, 200, 300, and 400 jobs per VO. Figure 4.13 captures the average required metric. It can be easily observed that the required metric increases more slowly with the number of jobs than with the number of sites. Thus, handling a single extra-job is less computationally intensive than handling an additional site.

Figure 4.13: Average Requirement Metric Function of the Workload Size

Figure 4.14 represents the variation of the average required metric for different numbers of VOs (10, 15 and 20) for three Grid environments (100, 150 and 200 sites). The total number of jobs in the system was constant, 1500 jobs, that were distributed evenly over the simulated VOs. I note that in the current GangSim implementation it is more expensive to add VOs to the simulated environment than increase the VO workload sizes (previous figure). This result is expected, because additional data structures must be maintained and processed for additional VOs.

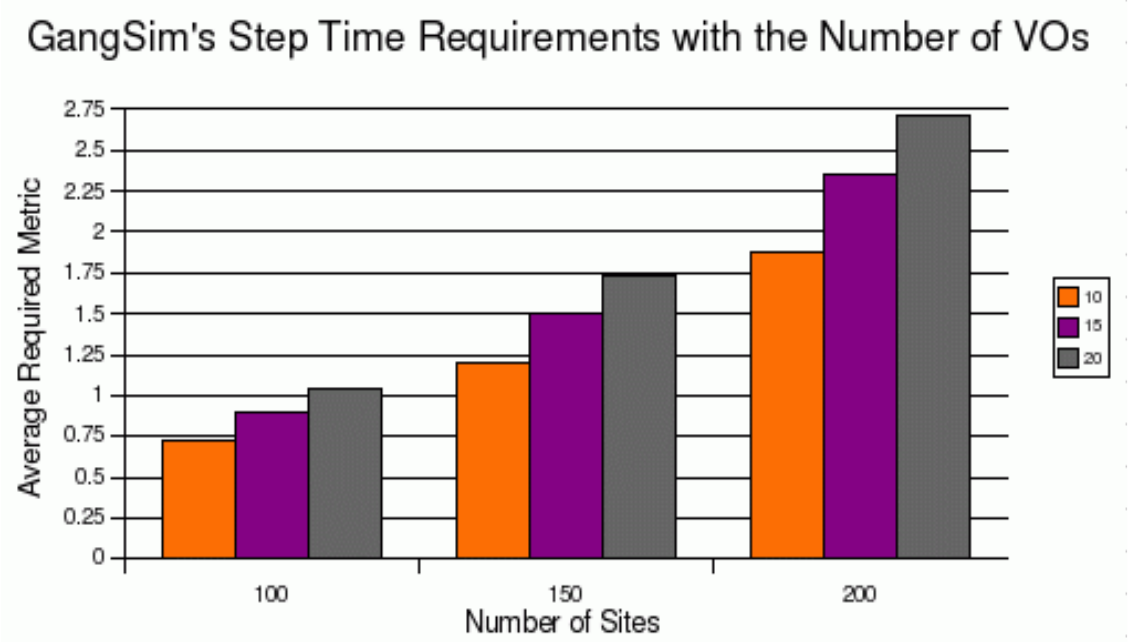


Figure 4.14: Average Requirement Metric with the Number of VOs for Three Environment Sizes

To summarize, adding either sites or VOs to the simulated Grid environment is computationally more expensive in the GangSim current implementation than increasing workload sizes.

4.4. Summary and Future Work

In summary, in this chapter I have detailed GangSim, a simulator developed for controlled resource sharing studies in Grid environments. I have also described the type of comparisons that can be made by means of a specific scenario. Other task assignment policies can be combined with various uSLAs for analyses. The simulation results show that commitment-limit uSLA performs the best for our sample scenario compared with the other three uSLAs introduced in Chapter 3 in terms of the computed performance metrics. GangSim has been used for several simulations studies and comparison studies regarding current Grid status [33, 38, 44, 107, 108].

As future work, I plan to in validate GangSim against real Grids. Preliminary simulator validation studies show inconclusive results. Grids introduce latencies and failures that are not captured by GangSim, which was developed as an ideal study environment. Second, a language-based uSLA specification represents an important step for extending GangSim's capabilities in simulating various scenarios. And third, GangSim can be configured to run in a distributed mode in which several simulator instances run on different hosts, with sites and computational load distributed appropriately. This feature has the potential to provide greater, but I have not yet evaluated whether this potential can be achieved in practice.

CHAPTER FIVE

GRUBER: A GRID uSLA-BASED BROKER

GRUBER is an infrastructure for uSLAs specification, management and enforcement in Grid environments. The current implementation has been successfully deployed, tested and used in the OSG/Grid3 environment [13]. GRUBER provides a means for uSLA discovery, management, and translation to allow Grid schedulers in order to perform uSLA-based scheduling. It addresses the issues of how uSLAs can be automatically discovered, retrieved, stored, and disseminated. The targeted entities are computing resources such as computers, storage, and networks, and computing services, such as Grid services.

The rest of this chapter is organized as follows. Section 2 presents the implementation details and enhancements required to handle large distributed Grid environments. Section 3 covers GRUBER's infrastructure performance. The chapter ends with my conclusions.

5.1 Implementation Details

GRUBER is composed of five software components (see Figure 5.1):

1. The *GRUBER engine* implements uSLA semantics, detects available resources, and maintains an overall view of resource utilization in the Grid. My implementation is a Grid service that is capable of serving multiple requests. It is based on the Globus Toolkit [85] and provides authentication, authorization, state or state-less interaction, etc.
2. The *uSLA language parser* provides support for translating the two specification languages introduced in Chapter 3 to an internal representation for the engine;
3. The *GRUBER site monitors and uSLA enforcers (S-PEP)* are the data providers for the GRUBER engine. They are composed of a set of UNIX and Globus tools for collecting Grid status information and enforcing uSLAs for resource managers that do not have means for controlled resource sharing;
4. The *GRUBER queue manager (QM) and VO uSLA enforcers (V-PEP)* are complex GRUBER clients that reside on submitting hosts. They enforce VO uSLAs and decide how many jobs to start and when;
5. The *GRUBER site selectors*, which reside on submitting hosts, are tools that communicate with the GRUBER engine to determine the best site for a job.

Currently, there are two implementations of GRUBER available, one based on Grid Services (OGSI [109]), and one on Web Services (WS [85]) - the two main versions of the Globus Toolkit (GT3 and GT4, respectively).

In this infrastructure, components exchange information under the following rules. The GRUBER engine periodically receives information about available resources on the Grid by means of the GRUBER SiteMonitor (Figure 5.1 - arrows 1 and 2). The

GRUBER SiteMonitor collects raw data from each local resource manager and translates it into the semantics of Chapter 3. Jobs are submitted to the local queue at each submission site and the GRUBER QM instructs the external scheduler (ES: Figure 5.1 - arrow 4), based interactions with the GRUBER engine (Figure 5.1 - arrow 3), when and how jobs can be released. As soon jobs are released, the ES interacts with one of the GRUBER SiteSelectors for selecting a specific site for each individual job (Figure 5.1 - arrow 5). A SiteSelector can be selected for each individual job. The SiteSelector uses information provided by the GRUBER engine (Figure 5.1 - arrow 6). The last step is the job submission (Figure 5.1 - arrow 7).

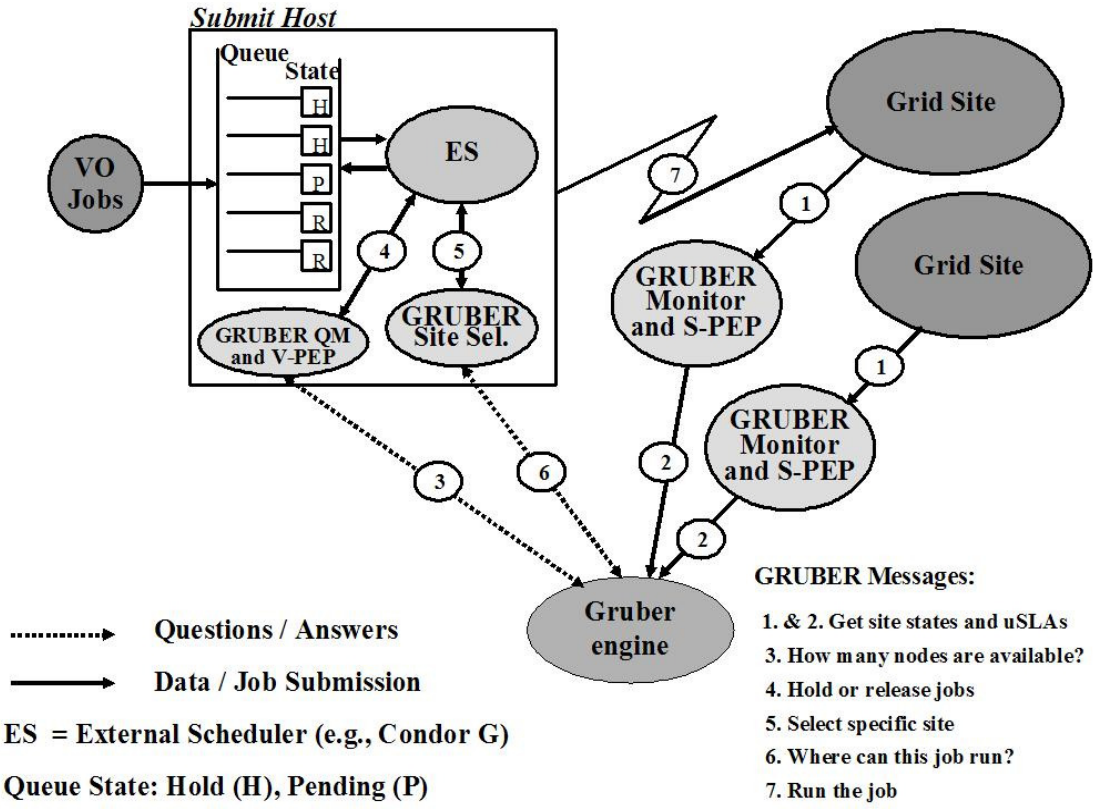


Figure 5.1. GRUBER Resource Brokering

5.1.1 The GRUBER Engine

The GRUBER engine represents the main component of the brokering infrastructure. I use the term *decision point* (DP) for an engine instance. A DP maintains a view of the available resources at each Grid site and invokes one of the four uSLA algorithms described in Chapter 3 (**no-uSLA**, **fixed-uSLA**, **extensible-uSLA**, and **commitment-uSLA**) for determining resource availability. All the other components in the brokering infrastructure communicate with a DP to perform their operations.

GRUBER decides which providers are available for a request using monitoring data and published uSLAs. The GRUBER engine's logic for managing CPU and disk requests is captured by the following algorithm, which returns the set S of available sites for use of job J, where J represents the job characteristics as well the job name, from the VO_i:

```
G    = Grid Site Set ;
J    = resource requirements for job J;
S    = Matching Site Set;

algorithm get-avail-sites_cpu inputs (G, VOi, J) returns S
1. S = empty
2. for each site s in G do
   # Apply one of the algorithms introduced in chapter 3
```

```
3.      if uSLA (s) == none and
                no-uSLA (J, VO_i, s) == accept then
4.         add (s, S)
5.      else if uSLA (s) == fixed and
                fixed-uSLA (J, VO_i, s) == accept then
6.         add (s, S)
7.      else if uSLA (s) == extensible and
                extensible-uSLA (J, VO_i, s) == accept then
8.         add (s, S)
9.      else if uSLA (s) == commitment and
                commitment-uSLA (J, VO_i, s) == accept then
10.         add (s, S)
11.     else
12.         next
13. return S
```

5.1.2 uSLA Enforcers and Observers

In this section, I describe two solutions for uSLA management and enforcement provided by GRUBER. The first solution considers the case of simple RMs that are unable to arbitrate among concurrent requests for resources, in which case complete policy enforcement points (PEPs) are required. The second solution supports advanced site RMs capable of enforcing complex uSLAs, in which case only policy observation points are required (POPs).

Solution 1 (Stand-alone resource S-PEP): The first solution works with any primitive batch system, for example Q^2ADPZ [110], that provides at least the following: accurate usage and hardware status information, basic job management (start, stop, held, remove operations), and job prioritization capabilities (increase, decrease operations). The S-PEP interacts with the RM(s). It continuously checks the status of all the jobs being managed by the RM and invokes management operations when required to enforce uSLAs. This functionality is accomplished by gathering site uSLAs as described in Section 3.5, collecting monitoring information about cluster usage from the local schedulers, computing CPU-usage parameters, and sending commands to schedulers to start, stop, restart, hold, and prioritize jobs. As an example, the processing logic of the S-PEP based on the commitment-uSLA is presented below:

EP_i = Epoch allocation policy for VO_i

BP_i = Burst allocation policy for VO_i

Q_i = set of queues with jobs from VO_i

BA_i = Burst Resource Allocation for VO_i

EA_i = Epoch Resource Allocation for VO_i

Utilization = current utilization on the site

TOTAL = possible allocation on the site

procedure *s-pep_commitment*

1. **while (true) do**
2. **sleep** 10 *# (seconds)*
3. **foreach** VO_i

 # Case 1: resource exhausted on this site

4. **if** EA_i > EP_i **then**
5. suspend jobs for VO_i from all Q_i

 # Case 2: available and BA_i < BP_i

6. else **if** Utilization < TOTAL

& BA_i < BP_i & Q_i has jobs **then**

7. start job J from some Q_i *# (e.g., FIFO sched)*

 # Case 3: resource contention: fill EP_i

8. **else if** Utilization == TOTAL

```
                & BA_i< EP_i & Q_i has jobs then
9.        if j exists & BA_j >= EP_j then
10.           suspend an over-quota job Q_j
11.       start job J from some Q_i # (e.g., FIFO sched)
```

For further clarification, BA or EA represents the share actually used by a VO. BP or EP represents upper limits for these shares. If BP or EP increases, then the VO receives additional resources. An important novelty of this approach over a cluster RM is its capability to keep track of jobs scheduled by several RMs on a single cluster, and to allow the specification of complex uSLAs without the need to change the RM implementations.

Solution 2 (Stand-alone resource S-POP): The second solution was developed and implemented with success in the context of the OSG/Grid3 environment. I decoupled the functionalities of the S-PEP by introducing a standalone *site policy observation point* (S-POP) and by allowing the RM to manage how local resources are shared. In this case, the assumption is that in addition to the functionalities enumerated for solution 1, the RM can enforce the desired uSLA. Examples of such cluster RMs are Condor [101], Portable Batch System [111], and Load Sharing Facility [103], which are widely used on OSG/Grid3 [31]. The S-POP translates to/from the RM usage policies to the common uSLA language described in Chapter 3, monitors resource utilization, and communicates this information to the GRUBER engine. This approach eliminates the requirement to deploy additional Grid elements for site management.

5.1.3 Queue Managers and VO-level uSLA Enforcement

GRUBER queue managers reside at the submission hosts and are responsible for determining how many jobs per VO or VO group can be scheduled at a certain moment in time. Usually, a VO planner is composed of a job queue, a scheduler, and job assignment and enforcement components. These last two components are part of GRUBER. A queue manager answers the question: "*How many jobs should group G_m of VO_n be allowed to run?*" and "*When can these jobs start?*" The queue manager is important for uSLA enforcement at the VO's level because it specifies, based on the VO uSLAs, how many jobs to run simultaneously for each VO's group. This mechanism also avoids site and resource overloading due to un-controlled submissions. The GRUBER queue manager implements the following algorithm (with the assumption that all jobs are held initially at the submission host):

```
J = Job Id ;
Q = Job Queue ;
S = Site Set ;
G = All Site Set ;
VO = Mapping Function jobId -> VO

procedure v-pep
1. while (true) do
   2.       sleep 10 # (seconds)
3.      if Q != empty then
```

```
4.          get J from Q
5.      else
6.          next
7.    S = get-avail-sites(G, Vo(J), J)
8.    if S != empty then
9.         release J from Q
```

5.1.4 Site Selectors

While GRUBER targets the provisioning of brokering services in Grid environments, I have also included scheduling mechanisms. By introducing site selectors, GRUBER can determine the *best* site for a running job in terms of uSLAs and current utilization. They are invoked directly by Grid schedulers (e.g., Euryale [100], KOALA [37], or WMS) in order to get site recommendations.

Currently, the four standard scheduling policies implemented by GRUBER are random selection (**G-RA**), round-robin selection (**G-RR**), most-recently-used selection (**G-MRU**), and least-used selection (**G-LU**). A fifth scheduling policy is provided by the *GRUBER observant* site selector (**G-Obs**), which associates a job with the site where the most recent job was started. In effect, this fifth site selector fills a site by assigning jobs to it until site's limit is reached.

5.1.5 Grid Service Brokering

GRUBER addresses resources and services differently, conditioned mainly by the local site managers in each case (i.e., Condor [112], PBS [102] for low-level resources,

ARESRAN [83], SAML [113] for services). For Grid services at the site level, GRUBER adopts the uSLA management approach from the ARESRAN prototype.

ARESRAN is a GT4 service for uSLA and reservation management, specification and enforcement at the level of a single site based on the Globus technology [85]. The ARESRAN prototype is based on the authorization schemes implemented by GT4, the so called *Policy Decision Point* (PDP). GT4 allows a chain of PDPs to be configured for each service, with each PDP evaluating to an independent decision [96]. The final answer is a logical AND operation of all these independent decisions. The ARESRAN authorization engine provides one of independent decisions to the GT4 authorization engine based on uSLAs enforced at the site level. I have developed two specific ARESRAN PDPs - one for managing service reservations and one for lower level resource reservations such as compute nodes that are managed by the WS-GRAM service [114].

The overall ARESRAN architecture is described in Figure 5.2. Whenever a service request arrives for a service managed through ARESRAN, the service's PDP steps in and verifies whether the request is acceptable. At the Grid level, GRUBER collects uSLAs from all individual ARESRAN services and provides brokering services for consumers.

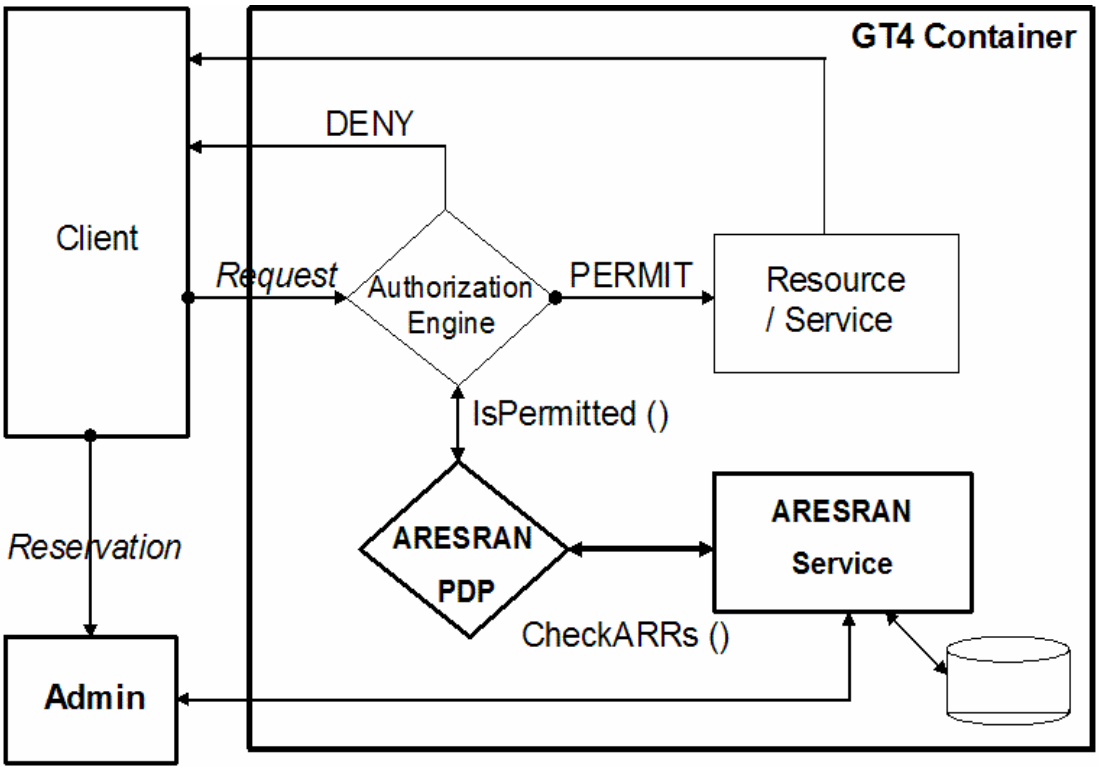


Figure 5.2. ARESRAN Architecture

The main components of these mechanisms are the ARESRAN Service, the ARESRAN PDPs and the ARESRAN Reservation Database. These three components communicate to ensure that requested resources or services are used appropriately. The specific details of these components are:

- **Service:** represents the reservation and uSLA engine of my prototype. Every time a new reservation is requested, the engine is invoked to verify whether the new reservation can be honored. The verification procedure uses information from the ARESRAN database and returns either ACCEPT, DENY or PROBABLY. If the reservation request is accepted, it is saved in the ARESRAN database;
- **Database:** stores reservations, uSLAs, and information about requests in progress. In this manner, ARESRAN has a complete view of the utilization of the services and resources that it manages. So far, the database is implemented in memory only, but future enhancements target the usage of a persistent database. Whenever a reservation is served, various statistics are also saved, such as: request time, running time, remote client, etc;
- **PDPs:** authorizes requests based on the rules stored in the database for various services and resources. Each PDP returns either *REJECT* or *ACCEPT*.

To accomplish service brokering, GRUBER collects the uSLAs enforced by each site and builds an overall view of the services' availability in the Grid, as is done in the resource brokering case. It collects current allocations and utilizations directly from ARESRAN (Figure 5.3 - arrow 2) and, based on this information, instructs the consumer about available alternatives (Figure 5.3 - arrow 3). GRUBER applies the

service brokering algorithm described in Chapter 3 to build the list of available alternatives for each individual client and service request.

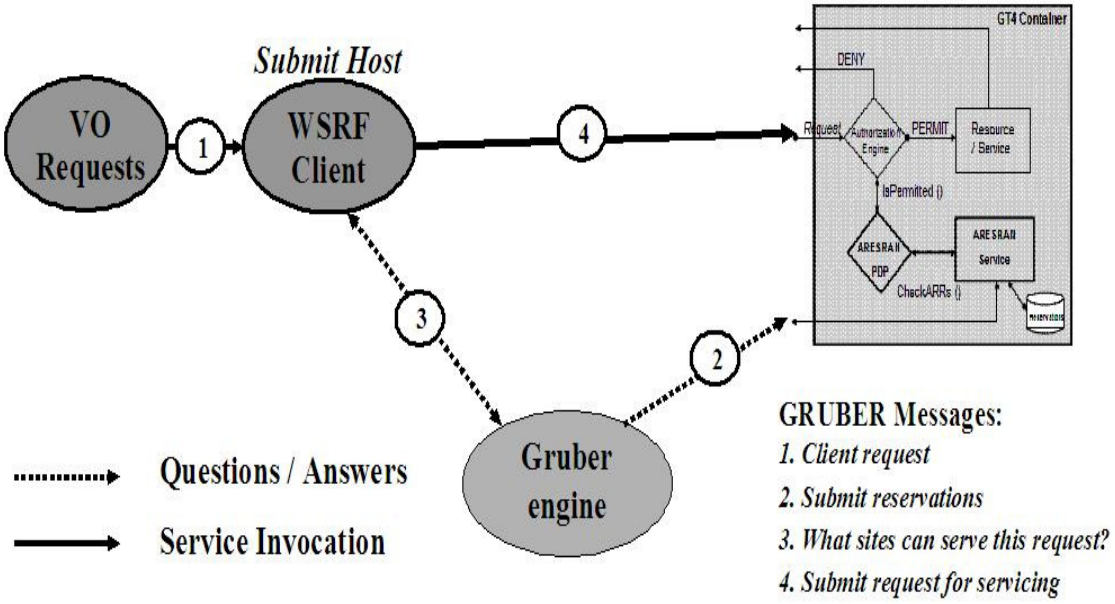


Figure 5.3. GRUBER Service Brokering

Because ARESRAN was developed from the beginning based on uSLAs for service sharing, the overall architecture is simpler and fewer component interactions are required. The complete GRUBER is presented in Figure 5.4.

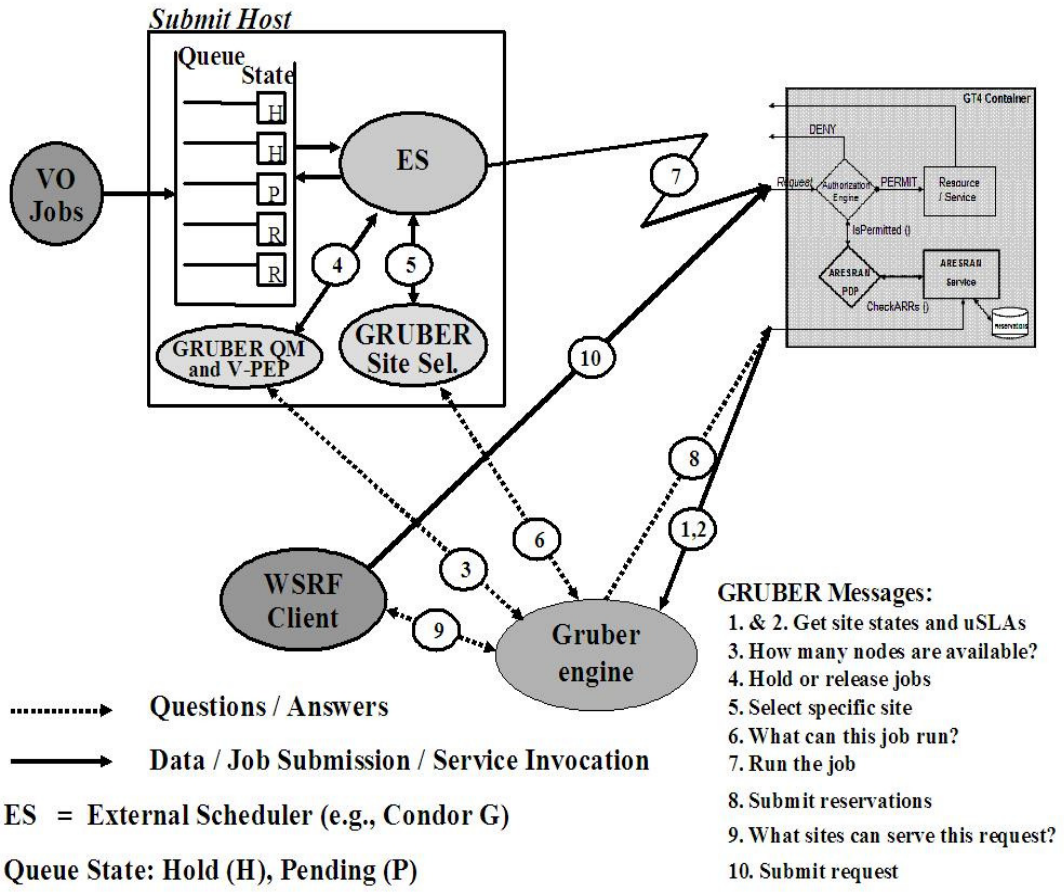


Figure 5.4. GRUBER Resource and Service Brokering

5.1.6 GRUBER Extensions

I implemented several extensions to GRUBER. The most important ones are described in this section: distributed uSLA management, multiple scheduling policies for associating clients with DPs, and a graphical verification interface for human operators.

5.1.6.1 DI-GRUBER (DIstributed GRUBER)

Managing uSLAs within environments that integrates participants and resources spanning many physical institutions is a challenging problem in practice. A single DP providing brokering decisions for hundreds to thousands of jobs and sites can become a bottleneck in terms of reliability as well as performance. I extended GRUBER with a distributed Grid uSLA-based resource broker, called DI-GRUBER that allows multiple DPs to coexist and cooperate in real-time.

DI-GRUBER aims to provide a scalable management service with the same functionality as GRUBER [45]. It is a two layer resource brokering service (see Figure 5.5), able to handle large Grid environments by extending GRUBER with support for multiple brokering DPs. The ability to integrate new DPs into an already existing brokering infrastructure is important in large and dynamic Grids.

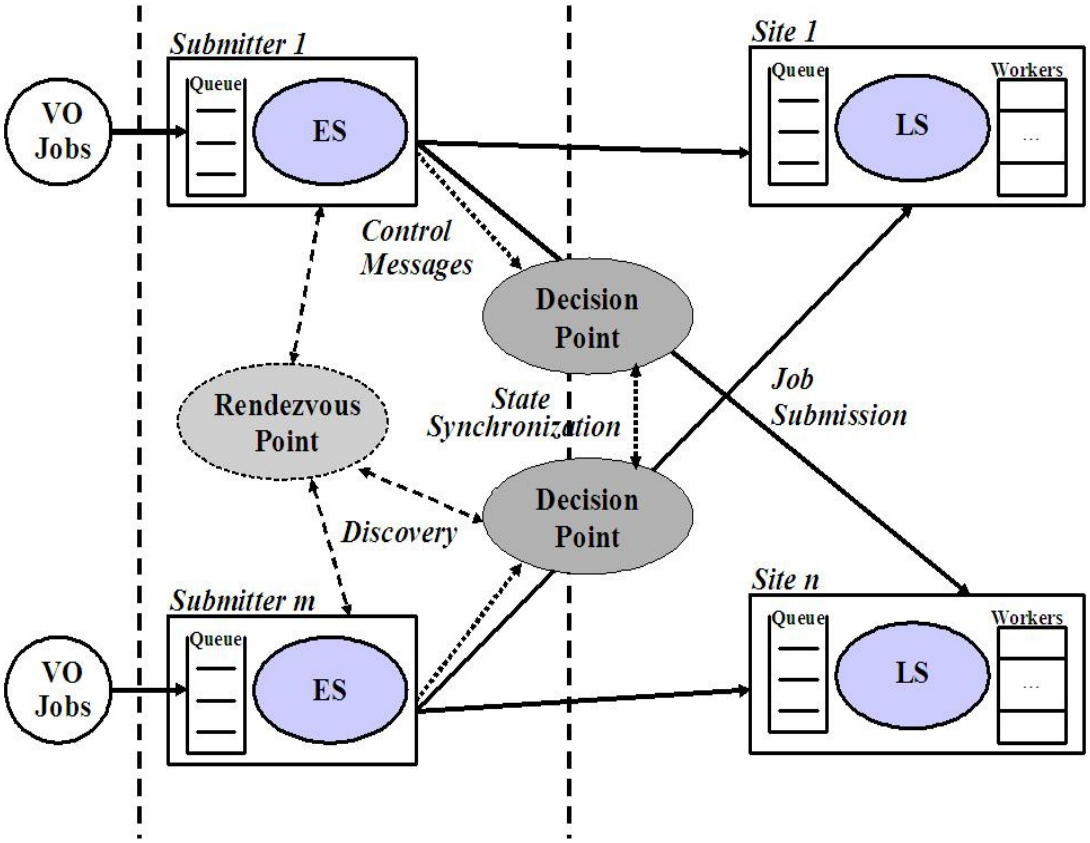


Figure 5.5. DI-GRUBER Architecture

DPs learn the brokering infrastructure based on a rendezvous mechanism. A human operator provides a rendezvous point and each DP queries and retrieves the list of the other DPs acting in the brokering mesh.

DI-GRUBER currently uses a rendezvous mechanism based on WS-Index Service [115] to integrate new DPs into the infrastructure. WS-Index Service [116] is a standard component of the Globus Toolkit that provides specialized functions for resource and service monitoring and discovery. WS-Index Service's main function in DI-GRUBER is to act as a specialized directory of all DI-GRUBER DPs for both clients and other DPs. Each DI-GRUBER DP registers with a predefined WS-Index Service at startup and is automatically deleted when it vanishes. Clients use this registry, based on a pre-defined scheduling policy, to select the most appropriate DP for usage. The scheduling policy could take into account metrics like load and number of clients already connected. So far, I have implemented the *least-used* policy based on the number of clients already connected to a DP. Figure 5.6 presents an example of the allocation of available DPs to the brokering clients and shows the method used for DP's address specifications (URI) and load (# of connected clients).

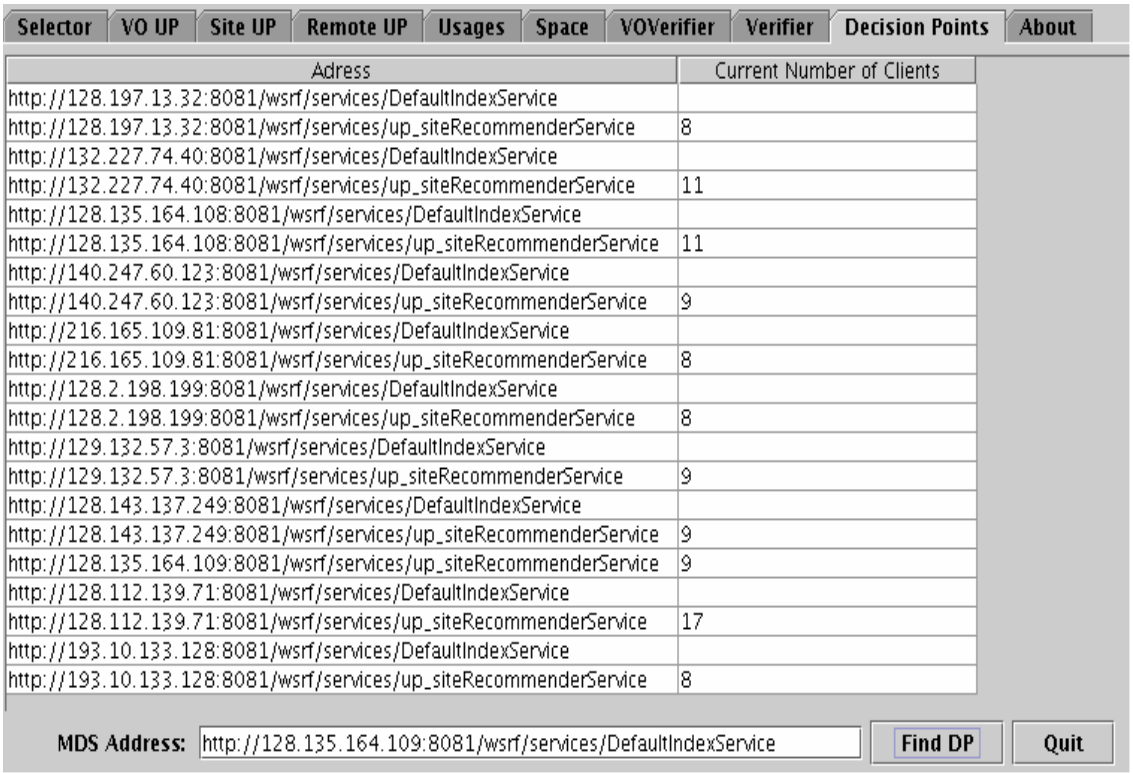


Figure 5.6. DPs Allocation Interface (PlanetLab experimental testbed)

When the infrastructure becomes overloaded, DI-GRUBER should start new DPs. Such dynamic bootstrapping is difficult to automate in a generic environment. The solution I have devised a semi-automatic method for the Grid3 scenario. When a client fails to communicate or to connect to a DP, it registers a request fault with the WS-Index Service. These faults are then used by a human operator to bring up new DI-GRUBER instances and re-stabilize the brokering infrastructure.

5.1.6.2 Verifiers

Monitoring of a brokering infrastructure is important in order to understand why certain decisions were performed and how the framework actually performs in different situations (the *verifier* concept introduced by Dumitrescu et al. [33]). As a first step towards this goal, I introduced a graphical-based verification mechanism that describes how resources are used by each client and presents the current allocations and uSLAs for each DP. This interface connects to DPs, collects the local or generic view of how resources are being managed, and presents it in an easy to visualize mode (see Figure 5.7).

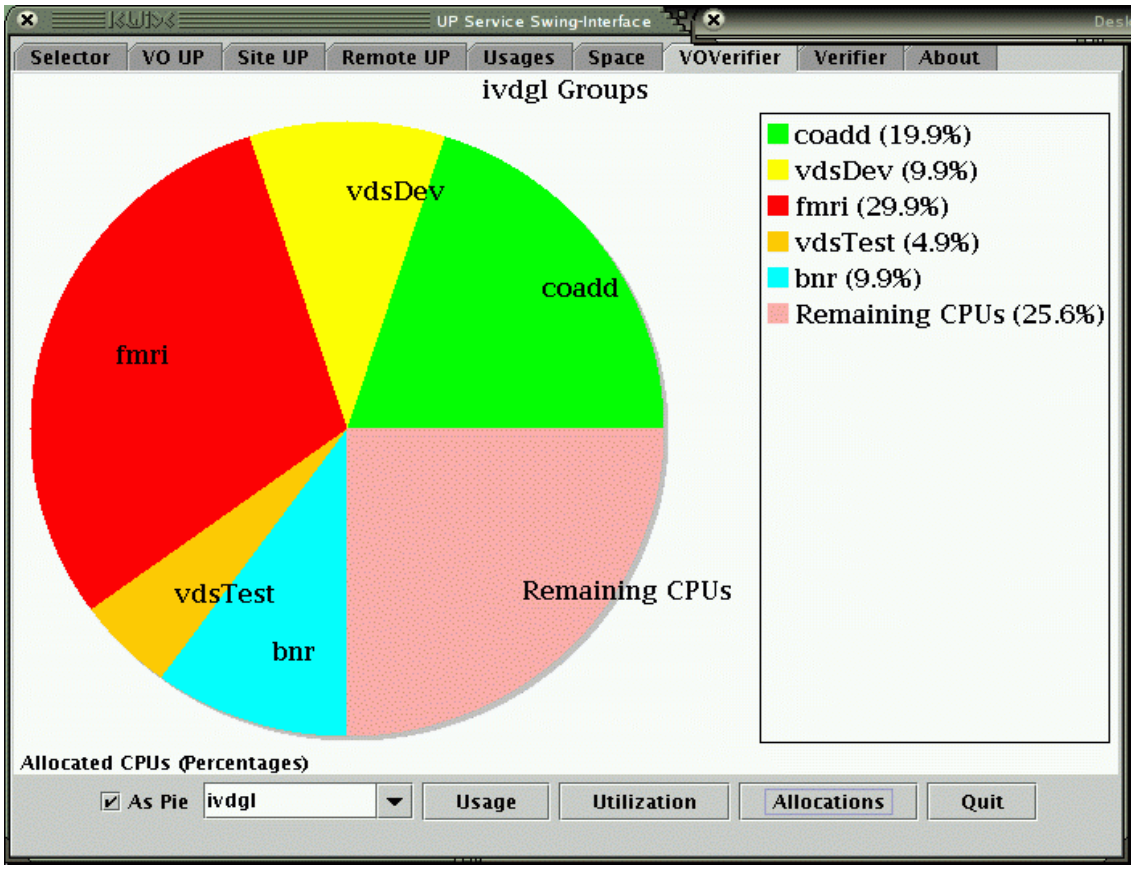


Figure 5.7. Resource Allocation Scenario

From a verifier point of view, the interface provides a means for addressing the following two questions: *"Are uSLAs adequately enforced by each DP?"* and *"What are the utilizations and allocations of different resource in the Grid?"*

5.2 The Performance of GRUBER

In this section, I focus on experimental results that illustrate GRUBER's capabilities in terms of both scalability and brokering accuracy. These results evaluate the performance of both the centralized and distributed versions of GRUBER.

5.2.1 Experimental Setup

In this section, I introduce my experimental setup and the performance metrics used for analysis. For each scenario, I describe the setup and workloads used to perform the experiment.

All the experiments described in this chapter were performed using the DiPerF performance testing framework [117]. DiPerF coordinates several machines in executing a performance testing client and collects metrics for the performance of the tested service. The framework is composed of a controller/collector, several submitter modules, and a tester component. In my experiments, testers run DI-GRUBER clients against the brokering infrastructure. Each such tester is launched with a predefined time interval between consecutive testers – 25 seconds for the experiments described in this chapter.

DiPerF collects metrics, described in section 5.1.2, as a function of either test execution time or the number of running testers. The advantages of using DiPerF are its capacity to coordinate large scale distributed tests involving 500+ clients and its automated performance metric collection.

I used PlanetLab as the testbed for all the experiments of this chapter. PlanetLab nodes are RedHat-based PCs connected to the PlanetLab overlay network with worldwide distribution. They are connected via 10 Mb/s network links (with 100Mb/s on some nodes), and have processor speeds exceeding 1.0 GHz (IA32 PIII class processor), and at least 512 MB RAM.

Each experiment consisted of a set of clients requiring brokering decisions from one, three or ten DPs. The DPs maintained a view of the entire Grid environment and periodically exchanged information with other DPs concerning recent job or request dispatch operations. Each experiment specifies its particular exchange period is specified for each experiment. DPs were connected in a mesh with various degrees of connectivity, as described for each experiment.

From 1 to 120 clients were used for performance testing. These clients ran on the PlanetLab nodes and each maintained a connection with one DI-GRUBER DP. Clients can select a DP under either a random or least-used scheduling policy, but for all the experiments in this chapter clients use the least-used policy. Each client was configured to apply a 60s timeout to the requests that it dispatched to a DI-GRUBER DP. If the timeout occurs, then the client's site selector recommends a site according to the local scheduling policy without considering site uSLAs. This requirement is introduced by

the need to provide a brokering decision in a predefined time interval. The Euryale scheduler [100] used for job submission on OSG/Grid3 had a fixed timeout equal to 60s for brokering decisions. The experiments were one hour in all cases. Clients submitted jobs once per second.

For the scalability and brokering accuracy experiments, the DPs maintained a complete view of an emulated Grid that was composed of 300 sites with a total of 40,000 nodes (a Grid approximately ten times larger than OSG/Grid3 today). The clients simulated brokering requests for 60 VOs and ten groups. Each GRUBER client randomly chose a VO id and group id under which the request was made. This selection process was repeated for each additional request.

The emulated Grid configuration was based on OSG/Grid3 configuration settings (spring of 2005) in terms of CPU counts, network connectivity, etc. The uSLAs also were derived from OSG/Grid3 settings: no-limit is used at the VOs' level, 50 sites share resources under commitment-limit, 170 sites share under extensible-limit, 70 under fixed-limit, and 10 under no-limit.

I note that this emulated environment is already as big as some existing P2P networks. There are two layers of communication in this environment; the sites can be thought of as super-nodes from a P2P network, while the resource nodes can be thought of as leaves from the P2P network.

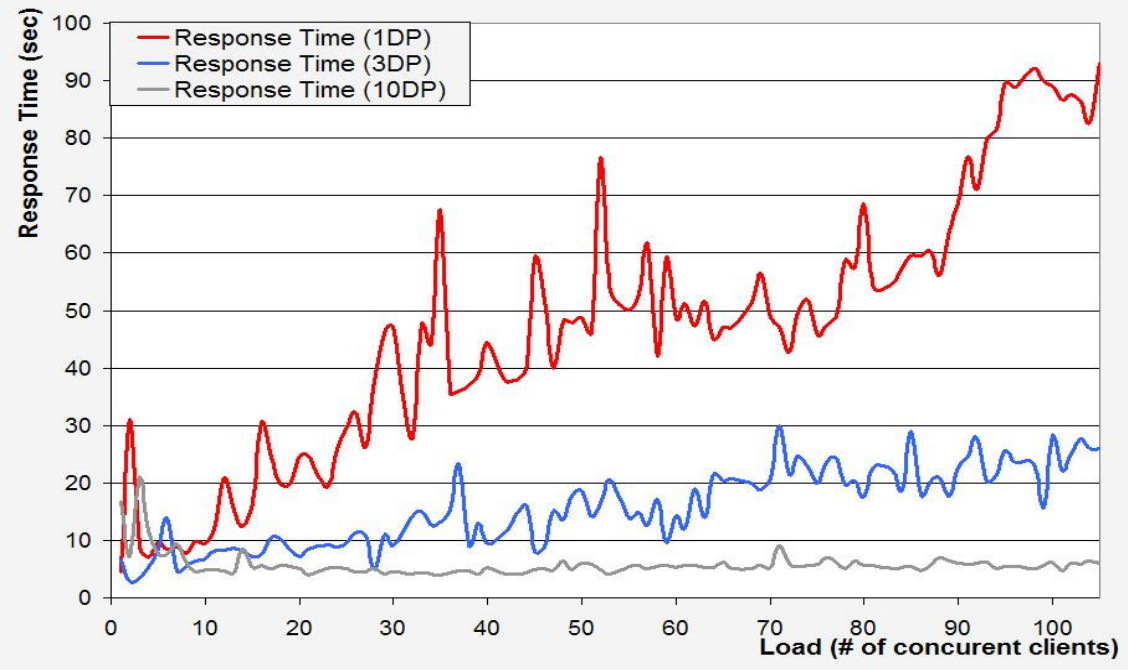
For the service brokering scenario, the environment was composed of ten ARESRAN-managed WSRF services deployed on PlanetLab nodes and the DPs provided brokering services for this ad-hoc service-oriented Grid. Each service ran

inside a GT4 container on a PlanetLab node, except DI-GRUBER and WS-Index Service, which ran inside the same container on a node at the University of Chicago having an Intel Pentium 2.0 GHz processor, 1 GB of memory, 100MBit/s network connectivity, and Linux-SuSe9.1 OS.

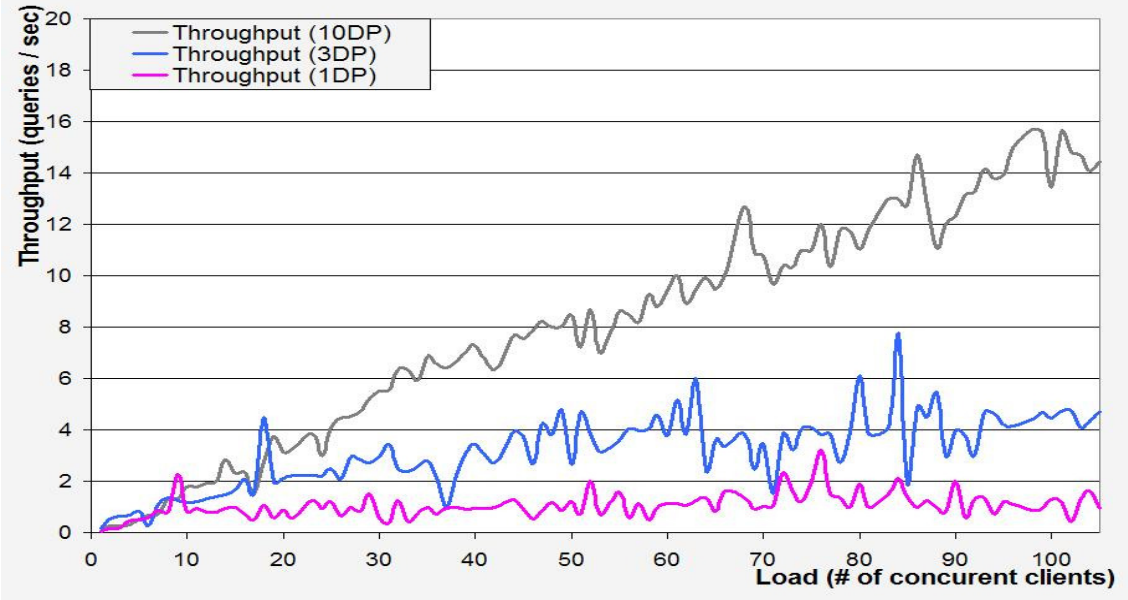
5.2.2 Scalability Test Results

Figure 5.8 reports the results for the experiments performed to measure DI-GRUBER scalability for one, three and ten DI-GRUBER DPs based on the emulated environment described in section 5.3.1. When three or ten DPs were used, they exchanged information about recently scheduled jobs every three minutes. The graph in Figure 5.8 plots the number of simultaneous active clients against the response time, while the graph in Figure 5.9 plots the number of simultaneously active clients against throughput.

As can be observed, the results show improvement in terms of *Throughput* and *Response Time* when moving from one DP to ten DPs. The *Throughput* metric's value increases almost linearly with the number of DPs, reaching a constant value of five queries per seconds for three DPs, while rising to 16 queries per second for ten DPs.



(A) Response Time Metric for one, three and ten DI-GRUBER DPs



(B) Throughput Metric for one, three, and ten DI-GRUBER DPs

Figure 5.8. DI-GRUBER Performance Metrics

The above results suggest that adding DPs increases performance, but they do not identify the number of DPs required to support a given level of performance. To evaluate this question, I extracted traces of the brokering requests made in the previous experiment. I also built the GRUB-SIM extension of DI-GRUBER, a trace-based simulator that is capable of simulating additional DPs for brokering requests, of identifying saturation moments, and of generating the optimal number of DPs needed for a given Grid environment. I used GRUB-SIM on these traces to compute the minimal number of DPs required to provide a 15 seconds response time for the considered environment.

The analysis is based on the **Response** metric and the moments when a client does not receive an answer in less than 15 seconds from the DP. It showed that a total of five DPs are sufficient to achieve a response time lower than 15 seconds for all 120 clients in the simulated environment.

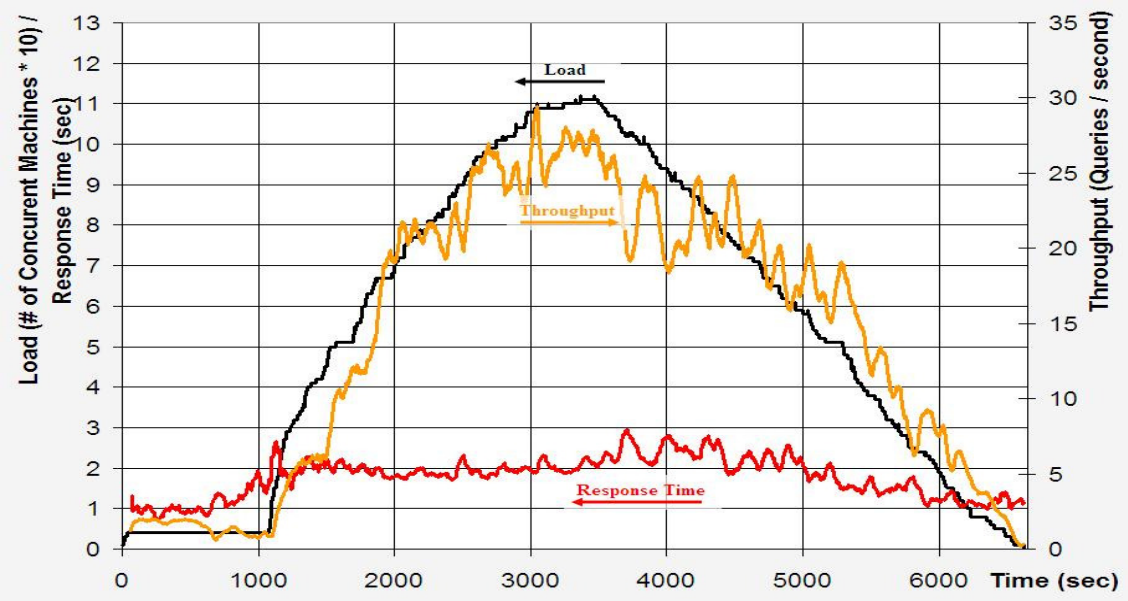
As a final note, the performance results presented above will remain un-changed for other uSLAs configurations. The four uSLAs have similar computation complexities for the same site: for a given brokering request the computational cost can be expressed as O (N+M), where N is the number of sites, and M the number of groups from a VO. If N is much larger than M, then the computational cost is just O (N). Even more, the cost introduced by communication surpasses the computational cost - to perform a request to a DI-GRUBER DP takes on the order of seconds to 10s of seconds, while the decision making operation is on the order of 100s of milliseconds.

5.2.3 Comparison with a Peer-to-Peer Service

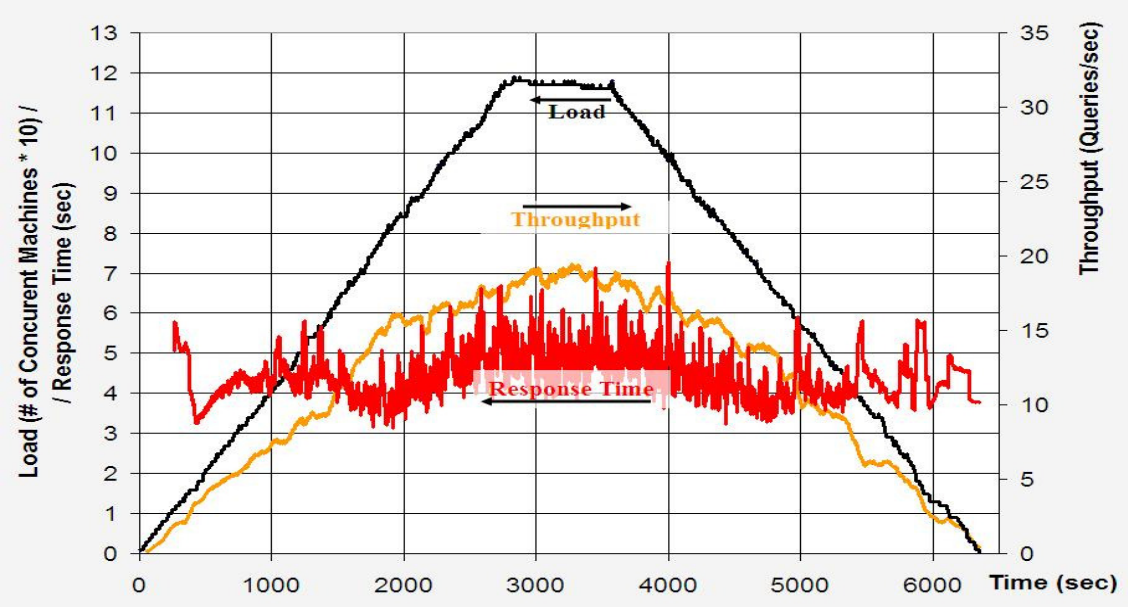
To evaluate DI-GRUBER's performance relative to other systems, I have studied the PAST application, built on top of the PASTRY substrate [8], using DiPerF on PlanetLab.

The setup used for the experiment was very similar to the one used for DI-GRUBER: five PlanetLab nodes for running permanent PAST nodes and 120 clients that joined and left the network in a controlled manner using the same 25 seconds delay. One PAST node played the role of the main rendezvous point (a node situated at UChicago). The remaining ones were maintained as backups. Each joining node requested a lookup and an insert operation every second (or, if the previous operation took more than one second, at soon as the previous operation ended).

The performance results are presented in Figure 5.9. In this case, the results are plotted as a function of the experiment execution time versus load/throughput and throughput. Figure 5.9 also includes results for ten DI-GRUBER DPs for easier comparison. They show that for insert and lookup operations, the PAST's response time is around 2.5 seconds (two to three times lower than for a ten DP instances of DI-GRUBER) with a higher variance in the beginning (the stabilization of the P2P network), while the throughput goes as up much as 27 transaction per second in average (1.6 higher than for DI-GRUBER). Finally, note that all operations were performed and measured on the local nodes (insertion followed by lookup); in the background the P2P network propagated the new elements in the network, which provides one partial explanation for DI-GRUBER's higher response time and lower throughput.



(A) PAST Network



(B) Ten DP DI-GRUBER

Figure 5.9. Response Time (left axis) and Throughput (right axis) for a variable Load (left axis * 10) for DI-GRUBER and PAST Network on 120 PlanetLab Nodes

5.2.4 Accuracy Performance Results

To evaluate the accuracy DI-GRUBER achieves, I consider three dimensions in the analysis space: mesh connectivity (defined as the number of peers used for exchanging data about recent brokering decisions), synchronization time interval among DPs, and the total number of DPs in the infrastructure.

The experiments are performed for each dimension by keeping two of the parameters constant, while performing the tests for different values of the third.

5.2.4.1 Accuracy with Mesh Connectivity

First, I measure **Accuracy** for brokering as a function of the DPs' average connectivity using ten DPs that exchange data every three minutes. I consider three cases: *full connectivity* (a DP sends its state to all the others), *half connectivity* (a DP sends its state only to half of the others), and *one-fourth connectivity* (a DP sends its state only to one quarter of all the others). Table 5.1 contains the results achieved by the DI-GRUBER infrastructure for configurations.

Table 5.1. Accuracy Function of the Infrastructure Mesh Connectivity

Connectivity (N=10)	Accuracy (%)
N-1	75
N/2	62
N/4	55

I note that the accuracy of the brokering infrastructure drops substantially as the connectivity decreases.

5.2.4.2 Accuracy with Exchange Time Intervals

Second, I measure the accuracy as a function of exchange intervals. I use three and ten DPs fully connected that exchange information every one, three, ten, and 30 minutes. The results in Table 5.2 show that, for a three DP infrastructure, a three to ten minutes exchange interval is sufficient for achieving **Accuracy** over 85%, while one to three minutes is required for achieving over 75% **Accuracy** for ten DPs. However, this accuracy value depends also on the number of the jobs scheduled by the DPs, but one job/s considered is sufficiently high for any Grid environment [13, 40, 80].

Table 5.2. Accuracy Function of the Exchange Time Interval for Three and Ten DPs

Exchange Interval (mins)	Accuracy for three DPs (%)	Accuracy for ten DPs (%)
1	89	80
3	87	75
10	86	68
30	83	61

5.2.4.3 Accuracy with the Number of Decision Points

Third, I analyze the performance of DI-GRUBER and its strategies for providing accurate scheduling decisions as a function of the number of DPs in the infrastructure. I use one, three, five and ten DPs that exchange data every three minutes under a full mesh connectivity. Table 5.3 depicts the accuracy performance and I note that the accuracy drops to 84% in the five DP case and 75% in the ten DP case, thus, having fewer DPs yields better accuracy.

Table 5.3. Accuracy Function of the Number of DPs

Number of DPs	Accuracy (%)
1	98
3	89
5	84
10	75

5.2.5 Service Brokering Example on an Ad-hoc Grid

In this section I focus on experiments for measuring service brokering performance. I consider three scheduling strategies, GRUBER, Round-Robin, and Random and two resource availability scenarios, fully available (FA) and partially available (PA). By partially available I mean that only two out of ten sites had resources available for consumption. The analysis is based on the **Response** metric, with the following

refinements: **GRUBER Response** represents the **Response** metric for GRUBER, **Service Response** represents the **Response** metric for the tested service, and **Reject Response** represents the **Response** metric for the authorization mechanism when a request is rejected due to uSLA constraints.

The fully available scenario represents the worst case performance for GRUBER. This scenario will favor simple scheduling approaches, such as the random or round robin assignment, because of the abundance of available resources. Furthermore, this scenario also shows the overhead incurred by using GRUBER in relation to the simpler assignments.

The partially available scenario represents the best situation for the GRUBER engine. This scenario will favor a scheduling approach that can make good and informed scheduling decisions. Table 5.4 depicts the results for clients situated on the same network as the GRUBER engine using two different scenarios for each scheduling strategy.

For the fully available scenario, I first observe that the total number of request completed in one hour differs greatly between the first third scheduling strategies (177 vs. 321 and, respectively, 312). As expected, these results show that brokering requests take up a significant fraction of the total execution time without an improvement in the assignments.

The partially available scenario demonstrates the potential benefits of GRUBER, which was able to obtain many more assignments than the random or round robin assignment strategies (150 vs. 59 and 52). I conclude that the utility of the GRUBER

service brokering engine occurs only when the number of services is large, custom advance reservations are performed in the system or the number of the requests is large and can potentially exhaust the available allocations.

Table 5.4. Service Brokering Performance Results (Metrics: Number of Request, GRUBER infrastructure Response time, Tested Service Response Time and Tested Service Reject Time)

Scheduling Strategy	GRUBER Assg (FA)	GRUBER Assg (PA)	Random Assg (FA)	Random Assg (PA)	Round Robin (FA)	Round Robin (PA)
# of Request	177	150	321	59	312	52
GRUBER Resp.	8.98	9.78	0	0	0	0
Service Resp.	11.45	16.55	10.84	17.45	11.04	17.06
Reject Resp.	0	0	8.89	10.18	9.28	9.85

5.3 Summary

In this chapter I presented GRUBER, a Grid resource broker, capable of uSLA resource management in a multi-site, multi-VO environment. It supports the uSLA-based Grid management infrastructure required by the scenarios presented in Chapter 3. I note that GRUBER is a complex service: a query to a DP may include multiple message exchanges between the submitting client and the DP, and multiple message exchanges between the DPs and the job manager in the Grid environment.

Managing uSLAs within large virtual organizations that integrate participants and resources spanning multiple physical institutions is a challenging problem. Maintaining a single unified DP for uSLA management is a problem that arises when many users and sites need to be managed. I provide a solution, namely the GRUBER infrastructure and the distributed version, to address the question on how uSLAs can be stored, retrieved and disseminated efficiently in a large distributed environment.

In summary, while GRUBER's performance is sufficient for today's Grids, the increase in scale of these environments will require scalable solutions [118]. DI-GRUBER is, in my view, such a scalable solution, powerful enough to handle world-scale Grids.

As an additional note, LCG [32] currently uses a solution based on several schedulers that rely solely on monitoring information gathered from provisioning sites. I believe that by incorporating some of the mechanisms developed for DI-GRUBER, the performance of LCG's scheduling infrastructure will increase substantially [80].

CHAPTER SIX

USAGE SERVICE LEVEL AGREEMENT RESOURCE MANAGEMENT

In this chapter I present experimental results on uSLA-based resource management. The main purpose of these experiments is to show that uSLA-based management can be achieved with success in real Grid environments. The results are grouped into three sections: the first set demonstrates that uSLAs are already enforced at the site level in Grids, the second set contains a comparison of the S-PEP and S-POP solutions proposed in Chapter 3, and the third set presents my evaluation of GRUBER, my solution for uSLA-based resource management on the OSG/Grid3 testbed. This last set of experiments also shows that GRUBER is scalable enough to support large workloads (at the limit of the other Grid technologies, *i.e.,* in this case Condor-G).

6.1. OSG/Grid3 Evolution

The Grid2003 Project has deployed a multi virtual-organization, application-driven grid laboratory (OSG/Grid3[1]) that, for almost two years, has sustained the production-level services required by physics experiments of the Large Hadron Collider at CERN (ATLAS and CMS), the Sloan Digital Sky Survey project, the gravitational wave search experiment LIGO, the BTeV experiment at FermiLab, as well as applications in molecular structure analysis and genome analysis, and computer science research projects in such areas as job and data scheduling. This infrastructure has been operating since November 2003 with 32 sites, a peak of 4500 processors, work loads from 10 different applications exceeding 1300 simultaneous jobs, and data transfers among sites of greater than 2 TB/day [31].

The infrastructure has evolved continuously during this interval to provide better services to its consumers. Also, resource providers joined constantly in the beginning. The software stack that supports this infrastructure, the Virtual Data Toolkit (VDT [119]), had a new release almost every two or three months. The VDT release documentation [119] contains a complete list of the system's software.

My experiments used the Virtual Data System (VDS [120]), which provides a means to describe a desired data product and to produce it in the Grid environment. The VDS provides a catalog that can be used by application environments to describe a set

[1] The first version of this environment was called WorldGrid.

of application programs (*transformations*), and then track all the data files produced by executing those applications (*derivations*).

6.2. Site-level uSLA Verification on OSG/Grid3

In order to verify that uSLAs are not only specified, but also enforced at the site level on OSG/Grid3's sites, I monitored site CPU utilization over a period of two weeks during *July 2004* at the University of Chicago site.

6.2.1. Monitored Configuration

The resource pool was managed by means of Condor, but other OSG/Grid3 RMs (e.g., OpenPBS [102], LSF [102, 103]) behave similarly [121]. Two VOs, USATLAS and GridExerciser, competed for resources on this site. This site's CPU utilization is captured in Figure 6.1. The uSLA enforced locally was **extensible-uSLA** with burst allocations of 35% for USATLAS and of 0.1% for GRIDEX, based on Condor's extensible fair share policy, and reported by the GRUBER's site monitor.

The software installed at the University of Chicago site was VDT 1.1.7 [119, 122], composed of Globus 2.0 [54, 123] for remote execution, and Condor-G 6.4.3 [124] – the software that supports workload submission to the Grid. The local site runs Condor 6.7.1 as site manager [20].

6.2.2. Results and Analysis

USATLAS's larger allocation means that whenever it has jobs queued locally, they acquire all resources. But as soon as the USTALAS load decreases (from time = 4th

day, 8th hour to time = 10th day, 20th hour), Grid-Exerciser's jobs take over and get all the resources they request. When the USATLAS jobs start to use all resources of the site again (X = end of 15th day), Grid-Exerciser's jobs are throttled back. Note that when USATLAS' load increases but does not fill the site's resources completely, the Grid-Exerciser jobs are not throttled back (from time = 6th day to time = 8th day and 16th hour).

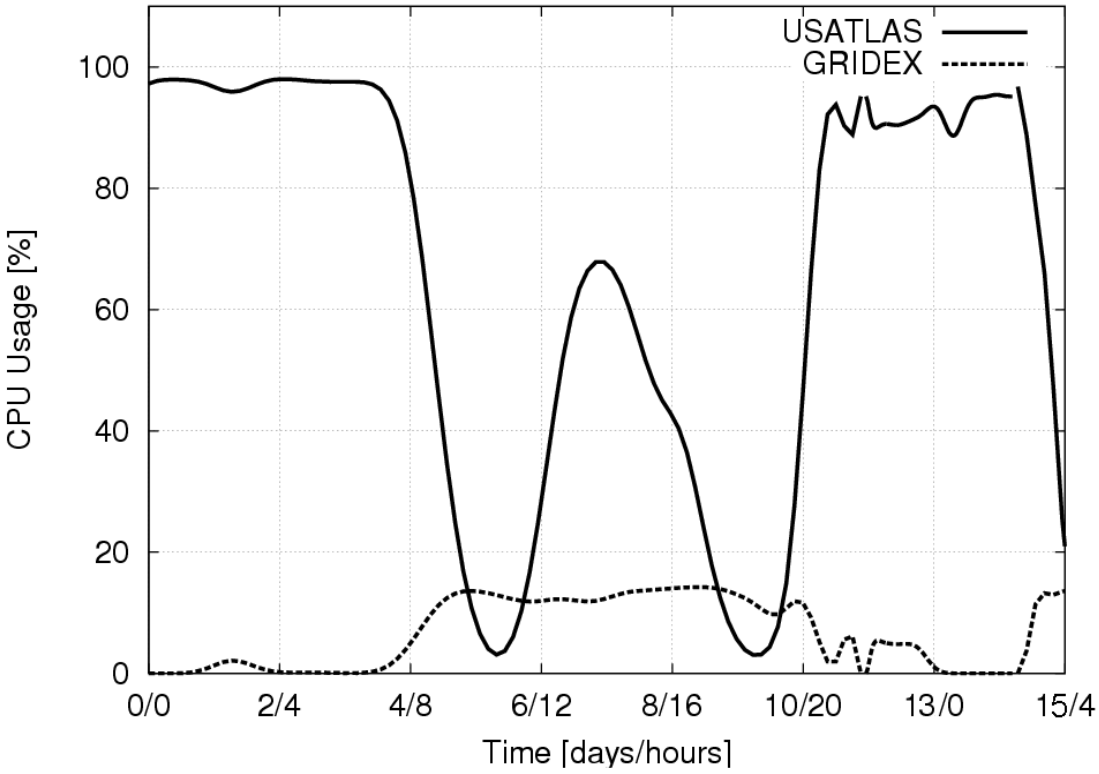


Figure 6.1. Resource Allocations at the University of Chicago's Site over an Interval of 15 days and 4 hours (time is expressed in days and hours)

6.3. S-PEP vs. S-POP Analysis

The main purpose of the experiments in this section is to compare the two solutions proposed in Chapter 3 for uSLA-based resource management at the site level. A *good* uSLA scheduler will maximize delivered resources and meet owner policies. Table 6.1 contains an example where resources are not allocated according to the local owner policies [33, 125].

Table 6.1. Possible uSLAs Scenarios for the VOs introduced in Chapter 3, Target represents the VO's burst limit, Current represents the VO's utilization, Demand, represents the VO's instantaneous request, and Level represents an uSLA violation indicator)

VO	Target	Current	Demand	Level
USCMS	60	50	50	OK
USATLAS	20	15	30	Under
IVDGL	10	10	100	OK
SDSS	5	22	50	Over

6.3.1. Synthetic Workload Description

I used synthetic workloads for this comparison. The two workloads overlay work entering a single site for two VOs, with the number of jobs and their average durations described in Table 6.2. Job arrival times and their durations have Poisson and Gaussian distributions, respectively. They mimic the workloads running on OSG/Grid3 [13], but

at a lower time scale (minutes instead of days) [121]. For each experiment, the execution period was 3000 seconds.

Table 6.2. Synthetic Workloads' Composition

VO ID	Number of Jobs	Average Job Duration [s]
0	400	250
1	480	200

6.3.2. Emulated Environment and Settings

This section describes the configurations used to perform the experiments. Two RMs are used for comparison: Condor [104] and Open-PBS [102, 126] in conjunction with Maui [126]. In each case, two VOs submit the workloads described above to a single 20-node site ($Site_0$) that is managed under either the S-PEP or S-POP solutions. CPU resources are allocated 20% to VO_0 and 80% to VO_1. The two VOs are allowed 30 second burst utilizations of 60% and 90% of the site's CPU resources, which can be expressed using the first syntax described in Chapter 3 as follows:

$< CPU, Site_0, VO_0, *, (3000, -20), (30, -60) >$

$< CPU, Site_0, VO_1, *, (3000, -80), (30, -90) >$

The experiments were performed during *January 2004*. Jobs were submitted via the Globus Toolkit 2.0 [3, 123], while the versions of the other components were Condor

6.6.3 [127], Open-PBS 2.3 [128], and Maui 1.0 [94, 129, 130]. The test site was monitored by collecting status information every ten seconds.

6.3.3. S-PEP-based uSLA Enforcement

The first set of experiments uses my S-PEP implementation, which performs uSLAs enforcement actions every 30 seconds. Figures 6.2-6.5 show instantaneous and total CPU utilization per VO as a function of time for the two VOs under the **commitment-uSLA**.

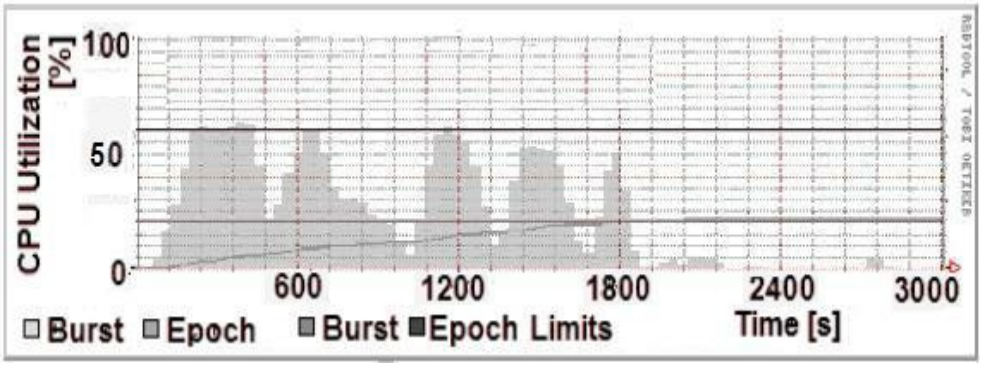


Figure 6.2. S-PEP with Condor (VO_0)

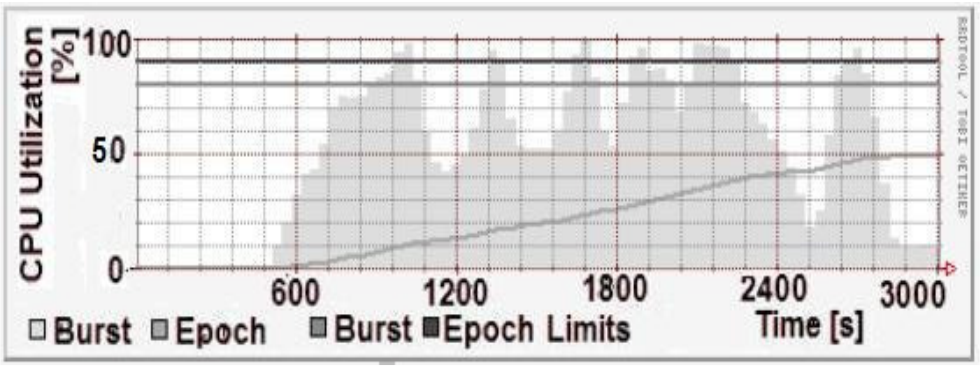


Figure 6.3. S-PEP with Condor (VO_1)

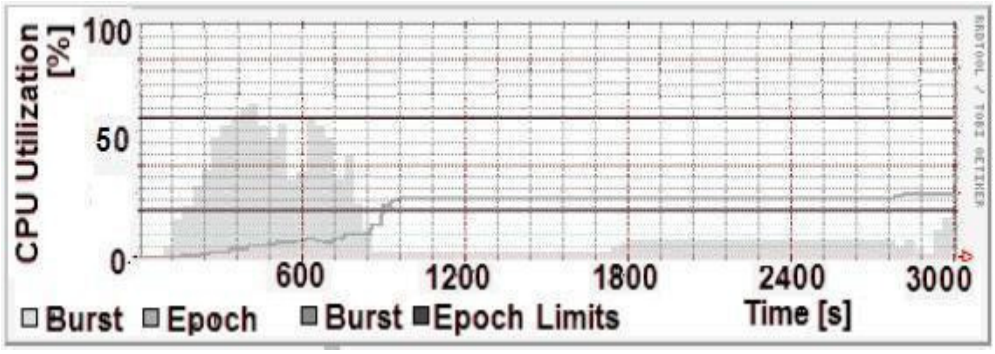


Figure 6.4. S-PEP with Maui/Open-PBS (VO_0)

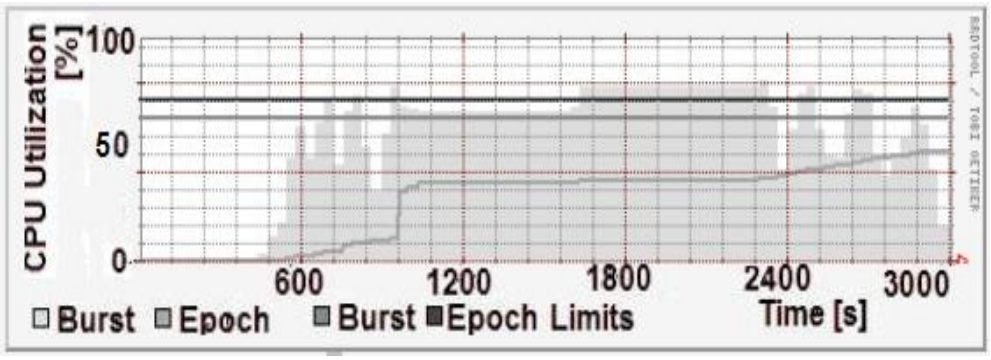


Figure 6.5. S-PEP with Maui/Open-PBS (VO_1)

I note that the uSLAs enforcement module has similar effects for both RMs. In the ideal case the burst allocation is never surpassed and resources are shared among the VOs according to their epoch allocation when contention occurs. This ideal behavior cannot be achieved due to latencies incurred in job control, and the subsequent

scheduling delay. In addition, the monitoring sub-component achieves different behaviors based on the capacity of the tested RMs to return job information in a timely fashion, for example, in Figures 6.4-6.5, from second 900 to second 1800, OpenPBS stops providing information due to system overloading; OpenPBS has higher computing requirements than Condor under the same testing scenario [129].

6.3.4. RM-based uSLA Enforcement

The second solution is entirely based on the local schedulers' ability to enforce uSLAs. uSLAs are specified as RM configuration rules, collected by the S-POP without any interference in the resource sharing process. Figures 6.6-6.9 show instantaneous and total CPU utilization per VO as a function of time for the two VOs and the two different local schedulers.

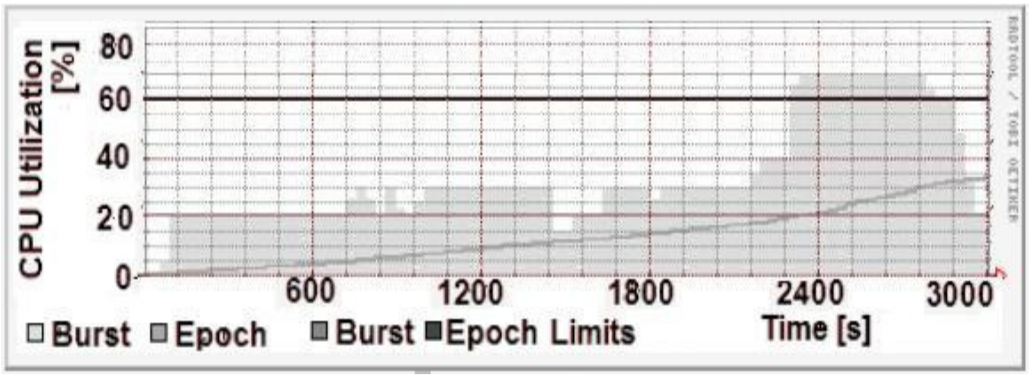


Figure 6.6. Condor as S-PEP (VO_0)

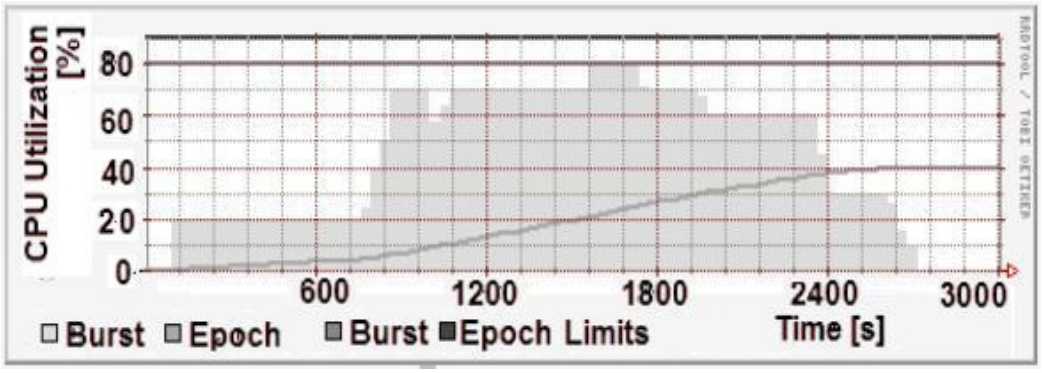


Figure 6.7. Condor as S-PEP (VO_1)

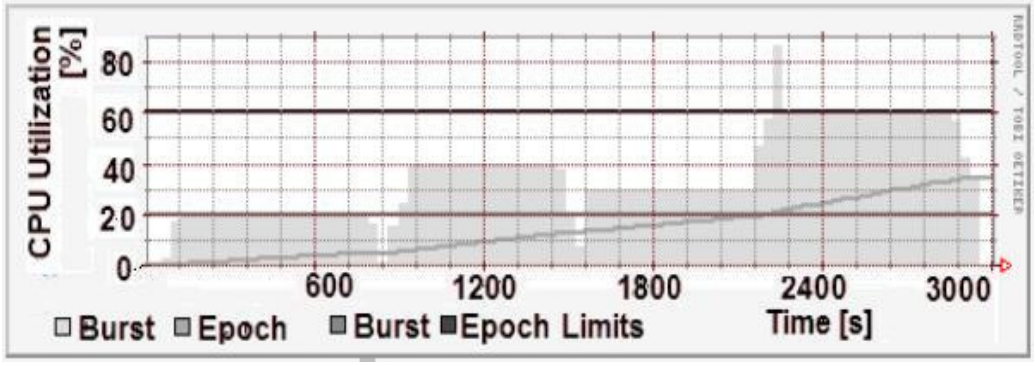


Figure 6.8. Open-PBS/Maui as S-PEP (VO_0)

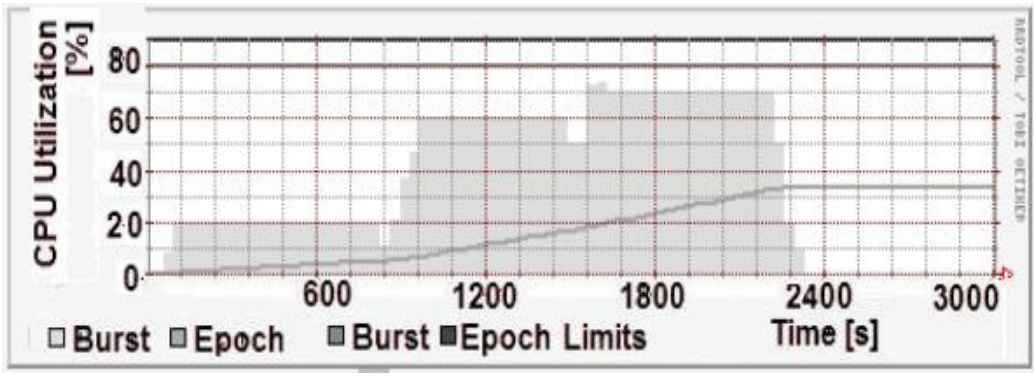


Figure 6.9. Open-PBS/Maui as S-PEP (VO_1)

For the S-POP solution, I observe a better balance of the jobs in execution by the RMs, thus the continuous utilization lines. The disadvantage of this approach is that it can only enforce the pre-programmed site uSLAs of each individual RM.

6.3.5. Quantitative comparisons

Finally, I present a quantitative comparison between the two solutions, which is captured in Table 6.3. I observe that for the scenarios considered, which are similar to the OSG/Grid3 environment presented later in this chapter, the S-PEP solution achieves better enforcement of an epoch uSLA than do the local resource managers; for the burst uSLA, there is no clear winner. Also, PBS/MAUI is better at enforcing burst uSLAs than Condor. The differences are within 10%, except the burst limit enforcement, where Open-PBS enforces the uSLA limits better.

Table 6.3. Quantitative Comparison Results

Configuration	Burst Accuracy (%)	Epoch Accuracy (%)	Util (%)
S-PEP/Condor	88	99	70.2
S-PEP/Open-PBS	90	92	88.5
S-POP/Condor	84	84	73.2
S-POP/Open-PBS	97	83	67.6

6.3.6. Conclusions

In this section I showed that uSLA-based resource management is possible at the site level by means of both a custom S-PEP solution and an S-POP based one. The experimental results show that in practice the S-POP alternative achieves a better balance of the execution of jobs, while the S-PEP solution offers similar behavior and provides uSLAs for any RM in general.

Also, while a stand-alone S-PEP alternative offers flexibility in uSLA specification and enforcement, it is not easy to develop a generic solution that interoperates with all available RMs in a real Grid deployment. In addition, the extra-burden on site administrators to deploy and learn such an S-PEP is not a viable alternative in practice [125]. In contrast, the S-POP approach has the advantage of being non-intrusive and overhead-free for a site administrator.

6.4. OSG/Grid3 Experiments

The experiments in this section show that, first, uSLA-based scheduling is possible in a real environment; second, GRUBER is scalable enough to support large workloads (1k to 10k - which were at the limit of the other technologies involved - e.g., Condor-G [101]); third, the performance achieved using GRUBER with its four site-selectors (both sequentially and parallel for the site-selectors for large workloads); and fourth, a comparison of GRUBER's brokering performance with two other approaches (**G-Observant** and R**andom**).

These two experiments used two types of workloads for evaluating uSLA-based management of Grid resources. First, large workload executions on OSG/Grid3 were performed to stress GRUBER performance on a real environment and to compare its performance with other available techniques. Second, small and medium workload executions on OSG/Grid3 are designed to measure the performance a user should expect when uSLA-based scheduling is employed. The experimental section also presents failure analysis based on the metrics collected during above experiments.

6.4.1. Workloads

For the OSG/Grid3 scheduling experiments, I used the bioinformatics sequence analysis program called BLAST [131]. A BLAST job, in my configuration, executes for 40 minutes on average, reads about 10-33 KBs of input, and generates about 0.7-1.5 MBs of output, that is, an insignificant amount of I/O.

I used this BLAST workload in three different configurations: (1) small workloads of 10, 50, and 100 jobs that are scheduled all at once; (2) medium and large workloads of 500 to 1000 jobs that are submitted in several steps according to the VOs and sites' uSLAs; and (3) very large workloads of 10k jobs that are placed in execution according to the sites' uSLAs. All workloads are composed of independent BLAST jobs. These workloads are of significant size; they are about $1/10^{th}$ the size of the traces analyzed by Iosup et al. in their work on four Grids around the world [80].

On a second dimension, I submitted the BLAST workload under two policies. In the *single-run-five-retries* case, I run a workload only once with a maximum of five retries

per job. In the *ten-runs-ten-retries* case, I submitted each workload ten times with a maximum of ten retries per job. A Grid environment introduces many points of failure due to the complexity of the technology involved, thus this final experiment offers a statistical view of what a user should expect a workload in such environments.

All the workloads were submitted under one iVDGL group, COADD, except for the case when the four BLAST workloads were submitted in parallel. In that case, workloads were submitted under BNR, COADD, FMRI and VDS-dev.

6.4.2. Configurations

This section describes the configurations used to perform the experiments on OSG/Grid3. I used 15 of 30 possible sites across the United States from the middle of *August 2004* through the end of *May 2005*, with the *single-run-five-retries* case ending at the end of *December 2004*, and the *ten-run-ten-retries* case starting in *January 2005*. The sites are autonomous, have heterogeneous resources (which explains the discrepancies between **Util** and **Time** metrics in a few cases), and are managed by different local resource managers, such as Condor [101], PBS or OpenPBS [94, 128], and LSF [105]. Each site enforces different uSLAs, which are collected by GRUBER's SiteMonitor and further used in scheduling workloads.

The software stack deployed during the experiments changed on a constant basis, due to the continuous improvements and bug fixes required by different partners. During the single-run-five-retries experiments reported in Section 6.4.3 VDT versions 1.2.2-1.2.4 were deployed and used on the OSG/Grid3 sites, while the window to the

Grid was VDS version 1.2.3. During the ten-runs-ten-retries experiments reported in Section 6.4.4 VDT versions 1.3.1-1.3.3 were deployed at the participating sites, while the window to the Grid was VDS version 1.3.4 [119, 120]. In all scenarios, GRUBER and Euryale were used as schedulers for running the workloads over OSG/Grid3.

Figure 6.10 presents a model of OSG/Grid3 and GRUBER's interactions [43, 87]. A single GRUBER DP is used for resource brokering by all the submission sites (from one to five as a function of the number of VO groups submitting workloads in parallel).

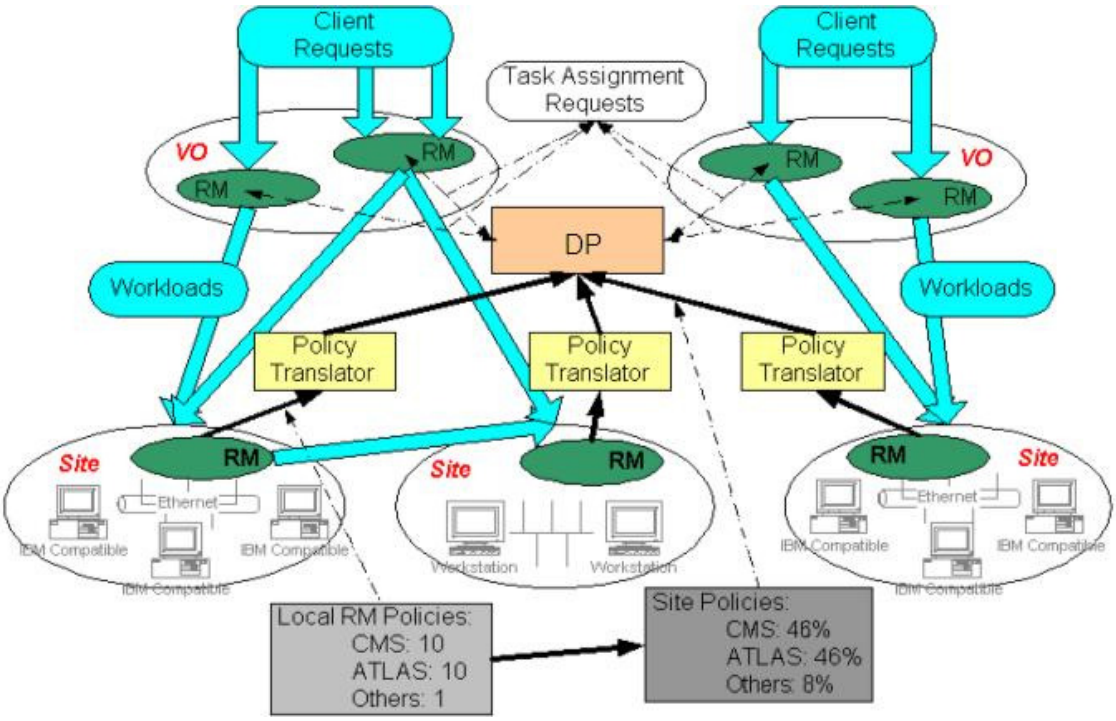


Figure 6.10. OSG/Grid3 Architecture

All jobs were submitted within the iVDGL VO, under specific VO uSLAs that allowed a maximum of 600 CPUs to be acquired.

For VO's, CPUs, storage and networks are the shared resources. Table 6.4 captures the VO CPU resource allocations in force on *July 9, 2004* on several sites in OSG/Grid3. The same uSLAs were in place for disk utilization, while network was provided by all sites under a **no-limit** uSLA.

Table 6.4. OSG/Grid3 Resource Sharing Example

Site Name	# of CPUs	Shares per VO (in %)		
		iVDGL	*USATLAS*	*USCMS*
t2cms0.sdsc.edu	76	(60, +1)	(60, +24)	(60, +1)
nest.phys.uwm.edu	305	(60, -0)	(60, +7)	(60, -0)
uscmst0.ucsd.edu	3	(60, +12)	(60, +12)	(60, +12)
xena.hamptonu.edu	1	(60, +25)	(60, +25)	(60, +25)
garlic.hep.wisc.edu	101	(60, +3)	(60, +3)	(60, +3)

For VO groups, the resources to be shared are virtual CPUs, storage, and networks, as provided and aggregated at the VO level. The uSLAs for these resources among VO groups were specified and enforced through GRUBER and were **no-limit** uSLA for storage and networks and as follows for CPUs:

```
< CPU, OSG/Grid3, IVDGL, *, –, (60, –20) >
< CPU, IVDGL, VDS, *, –, (60, –20) >
< CPU, IVDGL, FMRI, *, –, (60, –20) >
```

```
< CPU, IVDGL, BNR, *, –, (60, –20) >
< CPU, IVDGL, COADD, *, –, (60, –20) >
```

The re-planning interval was 20 minutes. A job is considered to have *failed* if it has been submitted five (*single-run-five-retries* case) or ten times (*ten-runs-ten-retries* case) unsuccessfully or if it is reported as having an application level "failure."

6.4.3. GRUBER-based Scheduling on OSG/Grid3

I present next, my results that show uSLA-based scheduling is possible and that GRUBER is scalable enough to support large workloads on OSG/Grid3. Four BLAST workloads of 1k and 10k jobs each were run sequentially under each of the four GRUBER's site recommenders (Tables 5 and 6) and then in parallel (Table 6.7).

Table 6.5. Performance Metrics for one 1k BLAST workload

Selector	G-RA	G-RR	G-MRU	G-LU		G-Observant	S-RA
Comp (%)	97	96.7	85.6	99.3		97.3	60.2
Replan	1396	1679	1440	1326		284	1501
Util (%)	12.85	12.28	10.63	14.56		13.59	0.57
Delay (s)	49.07	53.75	54.69	50.50		62.01	121.02
Time (s)	12044	7096	11461	11382		14073	32165

Sequential Site Selection Results: These results show that the various site scheduling policies offer different performance, and also provide a comparison with two non-uSLA enabled policies, G-Observant and S-RA. The total running time (**Time**)

was approximately five hours in each case. The **Re-plans** metric results are explained by GRUBER's approach for removing starved jobs after 20 minutes of starvation. In many cases jobs had to wait at slower responding or CPU-overloaded sites or wait for jobs running over the uSLA limit (under the extensible-limit uSLA) when pre-emption was not in place on the Grid3's sites. Also, some of the sites had software problems or insufficient resources on some of their nodes (see Table 6.9 for a detailed error list).

The total time for workload completion also varied under various site selectors. The best time was achieved for G-RR, **Time <** 7500s. Taking in account that a job runs to completion on average 40 minutes, the workload took 30% more than the ideal case when all jobs start and run as soon as slots are available.

Next, I compare GRUBER's performance in scheduling jobs with a basic random assignment technique and the observational approach in job scheduling. The **G-Observant** site selector submits jobs to a site as soon as the latest job sent to the same site was started. It fills up a site by sending jobs until site's quota is reached.

The results also show that **G-RA** achieves comparable performance in terms of time with the **G-Observant** site selector, while **G-RR** gets twice the performance of **G-Observant**. Compared with these two site selectors, the naive random assignment site selector, **S-RA**, performs two to three times worse in terms of the measured metrics. An important metric to observe is the number of re-planning operations. While the **G-Observant** site selector had around 300 such operations, **G-LU** performed around 1300 re-scheduling operations and the naive site selector around 1500. The explanation for the good performance of **G-Observant** on this metric is that a few sites scheduled jobs

much faster, and this site selector took also advantage of this characteristic in scheduling jobs.

Table 6.6. Performance Metrics one 10k BLAST Workload

Selector	G-RA	G-RR	G-MRU	G-LU
Comp (%)	91.75	91.88	73.58	77.88
Replan	18000	23900	24350	27718
Util (%)	2.43	1.32	1.76	3.31
Delay (s)	86.63	85.17	90.45	89.01
Time (s)	156437	152844	167092	127874

GRUBER's Scalability: To demonstrate GRUBER's scalability over OSG/Grid3, I used larger BLAST workloads composed of 10k jobs. These results are gathered in Table 6.6. Round-robin and random-assignment achieved the best performance in the case of 10k BLAST workload as well.

Concurrent Site Selection Results: While the above results are meaningful when only one workload is run under GRUBER's control, for this set of experiments I focus on measuring the scheduling performance of GRUBER's four site selectors when used in parallel for different workloads.

My results show that under higher resource contention, only **G-LU** does not perform as well as before in terms of **Time**, while GRUBER's the other three site selectors behave similarly relative one to another. Higher resource contention imposed by the four simultaneous workloads cause **G-LU** to prefer sites with fewer resources. In this

case, the **G-RR** and **G-RA** provide the best job completion rate with the smallest **Delay** and highest **Util**. Note that the **Time** metric has increased by a factor varying between 4.5 and 7.8 compared to the sequential case.

Table 6.7. Performance Metrics for four Concurrent 1k BLAST Workloads

Selector	G-RA	G-RR	G-MRU	G-LU
Comp (%)	98.7	98.2	87.9	87.9
Replan	1815	1789	1421	2409
Util (%)	14.02	13.51	11.05	11.52
Delay (s)	64.41	66.62	68.97	63.96
Time (s)	54735	55350	72275	64558

Failure Analysis: In this section, I discuss the sources of failure in the version of OSG/Grid3 used for this experiment. There are many cases in which a service may fail in a dynamic, heterogeneous, and large-scale environment. For example, failures may occur in a Grid at the infrastructure, middleware, application, and user levels, and may be transient or permanent. Due to the natural heterogeneity of Grids and their sheer size, failures appear much more often than in traditional parallel and distributed environments [80]. I analyze the most common errors I have faced in running large BLAST workloads. There are two types of failures: a *job failure* occurs when a job fails to run at a certain site, an error that results in job rescheduling, and *workload failure* occurs when under 100% of a workload completes.

A job error occurs when it failed to complete after a fixed number of retries. A failed job is not scheduled to the same site a second time, because Euryale tracks bad

sites for each job, but a different job could be. Euryale's re-submission operation keeps track of site failures only for the current job, but does not provide specific feedback to GRUBER about the execution result. Thus, the GRUBER engine does not have direct knowledge of site failures; instead, it traces differences between how many jobs run versus how many slots the site reports and tries to reconcile these numbers (what we call - *automated uSLA violation management*). The causes of workload failures are, in most cases, the small number of retries used during these tests and a few cases the DAGMan, a Condor-G tool for workflow management [132], crashed during workload management. The failure for the last try of the first failing job of a workload represents the workload failure error.

Job failures are due to the temporary failure of the Replica Location Service (RLS [133]) server used to stage data in and out (overloading issues), gatekeeper overloading and transient different authentication errors, transient RLS errors, etc. Table 6.8 captures error data for a sample of 15k BLAST job runs over OSG/Grid3.

Table 6.8. Error Percentages of 15k BLAST jobs submitted as 1k workloads (Percentages are computed as the ratio of current errors to the total number of errors)

Error Description	EPJ	Workload failure (#, %)	Job failure (#, %)
Remote Local Scheduling Timeout	0.68	3 (20)	10204 (29)
Remote Application Execution Failure	0.46	2 (13)	6846 (19)
Remote Transient Authentication Error	0.02	0 (0)	361 (1)
Transient Database Failure (RLS)	0.14	1 (6)	2158 (6)
Remote Unset Environment	0.82	4 (26)	13034 (38)
Remote Transient GridFTP Failure	0.02	5 (33)	1141 (3)
Remote GRAM Environment Error	0.034	0 (0)	526 (1)
Others	0.0	0 (0)	8 (0)
Totals	2.28	15 (100)	342768 (100)

6.4.4. Statistical Results for GRUBER Scheduling on OSG/Grid3

In this section I present the results for *ten-runs-ten-retries* performed using the same settings as before. While previous section proves GRUBER's scalability for scheduling large workloads, this section shows the performance of GRUBER for various small and medium workload sizes, and is intended to show the behavior a user could expect. Note, these experiments were started in *January* 2005, one month after the end of the experiments reported in Section 6.4.3 (*January 2005*), and most of the sources of failure identified during those experiments were fixed by upgrading to newer versions of the VDT [122] and VDS [100] software packages.

Small Workload Results: Table 6.9 shows the results for the 10 BLAST jobs. As can be seen, the speedup earlier sequential execution of the jobs is 2.5 to 3.5 smaller than the optimal due to the probability of a job ending on a site with a local resource manager that does not behave as expected (*i.e.*, has higher latencies for starting jobs, is overloaded due to numerous subsequent submissions, etc.). However, 75% of the jobs completed in a time interval closer to the ideal case.

Table 6.9. Average Results and 90% Confidence Intervals of Four GRUBER Strategies for a 10 job BLAST Workload (each workload was run 10 times and confidence intervals are based on these 10 runs)

Selector	Seq.	G-RA	G-RR	G-MRU	G-LU
Comp	100	100	100	100	100
Replan	0	34.1 ± 5.51	47.5 ± 9.26	13.6 ± 2.18	8.6 ± 1.83
Util (%)	0.15	0.36 ± 0.05	0.31 ± 0.07	0.50 ± 0.04	0.55± 0.10
Delay (s)	0	3262 ± 548	4351 ± 824	801 ± 313	1162 ± 376
Time (s)	28975	12436±1191.4	13966±2208.8	7653±1205	8787±158
Speedup	1	2.33 ± 0.25	2.21 ± 0.35	3.46 ± 0.45	3.6 ± 0.6
Spdup75	1	3.72 ± 0.59	3.46 ± 0.51	5.66 ± 0.55	5.32± 0.67

Table 6.10 shows the results for a workload with 50 BLAST jobs. As before, several jobs starved and their execution time affected the overall speedup. The speedup of 75% is more than the half of the ideal speedup (*i.e.*, 70% for the **G-RR**), showing that most jobs complete in close to the optimal time (37 of the total jobs).

Table 6.10. Average Results and 90% Confidence Intervals of Four GRUBER Strategies for 50 BLAST Workloads (each workload was run 50 times and confidence intervals are based on these 50 runs)

Selector	Seq.	G-RA	G-RR	G-MRU	G-LU
Comp	100	100	100	100	100
Replan	0	35 ± 14	51.1 ± 28	78.8 ± 9.51	48.8 ±10.8
Util (%)	0.07	1.18 ± 0.25	1.44 ± 0.27	1.76 ± 0.18	1.89 ±0.43
Delay (s)	0	1420 ± 713	583 ± 140.4	1260 ± 528.7	653.8 ±202
Time (s)	131372	8035 ± 990.4	9654 ± 603.5	9702 ±1247.3	8549 ±898
Speedup	1	16.35 ± 1.17	14.12 ± 0.90	12.76 ± 0.71	15.16±2.42
Spdup75	1	30.84 ± 5.70	35.36 ± 2.79	24.36 ± 2.28	35.41±2.48

Table 6.11 shows the results for a workload of 100 BLAST jobs. Again, similarly to the 50 BLAST jobs, the execution performance is half for 75% of the jobs and drops further for the entire workload.

Table 6.11. Average Results and 90% Confidence Intervals of Four GRUBER Strategies for 100 BLAST Workloads (each workload was run 100 times and confidence intervals are based on these 100 runs)

Selector	Seq.	G-RA	G-RR	G-MRU	G-LU
Comp	100	100	100	100	100
Replan	0	228.7 ± 21	39.9 ± 13.8	230 ± 20.3	124.7 ± 17
Util (%)	0.12	2.86 ± 0.30	3.48 ± 0.59	1.87 ± 0.46	3.51 ± 0.7
Delay (s)	0	1691 ± 198	529 ± 92.67	1244 ± 387.9	640 ± 93.4
Time (s)	232150	10350 ± 565.9	9013 ± 1025.1	7507±2325.1	9716±1130
Speedup	1	22.43 ± 1.55	30.15 ± 3.43	19.24 ± 1.56	28.02 ± 5.4
Spdup75	1	47.38 ± 3.24	77.19 ± 3.26	35.86 ± 3.72	73.54 ± 2.0

Medium Workload Results: Table 6.12 shows the results for the 500 BLAST jobs. The size of the workload allows the execution performance to increase. For example, 75% of the workload matches the ideal speedup and it is half for the overall workload.

Table 6.12. Average Results and 90% Confidence Intervals of Four GRUBER Strategies for 500 BLAST Workloads (each workload was run 500 times and confidence intervals are based on these 500 runs)

Selector	Seq.	G-RA	G-RR	G-MRU	G-LU
Comp	100	100	100	100	100
Replan	0	925 ±103.5	816 ±245.6	1024 ± 154.2	680 ± 139.3
Util (%)	0.50	34.04 ± 4.55	33.19 ± 2.39	25.41 ± 5.6	30.3 ± 4.7
Delay (s)	0	9202 ± 1716.8	6700 ± 816.6	9125 ± 6117.8	6169 ± 407
Time (s)	1892769	28116 ± 2881	24225 ±1035.9	20434 ± 4100	21362 1250
Speedup	1	67.32 ± 5.6	60.22 ± 3.26	51.77 ± 5.94	63.12 ±3.41
Spdup75	1	98.43 ± 8.7	111.69 ± 9.81	101.48 ± 10.05	113.2 ±8.82

Figure 6.11 presents the speedup performance over all runs with 90% confidence intervals. Note the very small confidence intervals, which expresses the low standard deviation, and hence the strength of my results across all the runs and all the configurations used in this section. I consider that these results to offer of a good prediction of the performance that can be achieved for a Grid environment like OSG/Grid3.

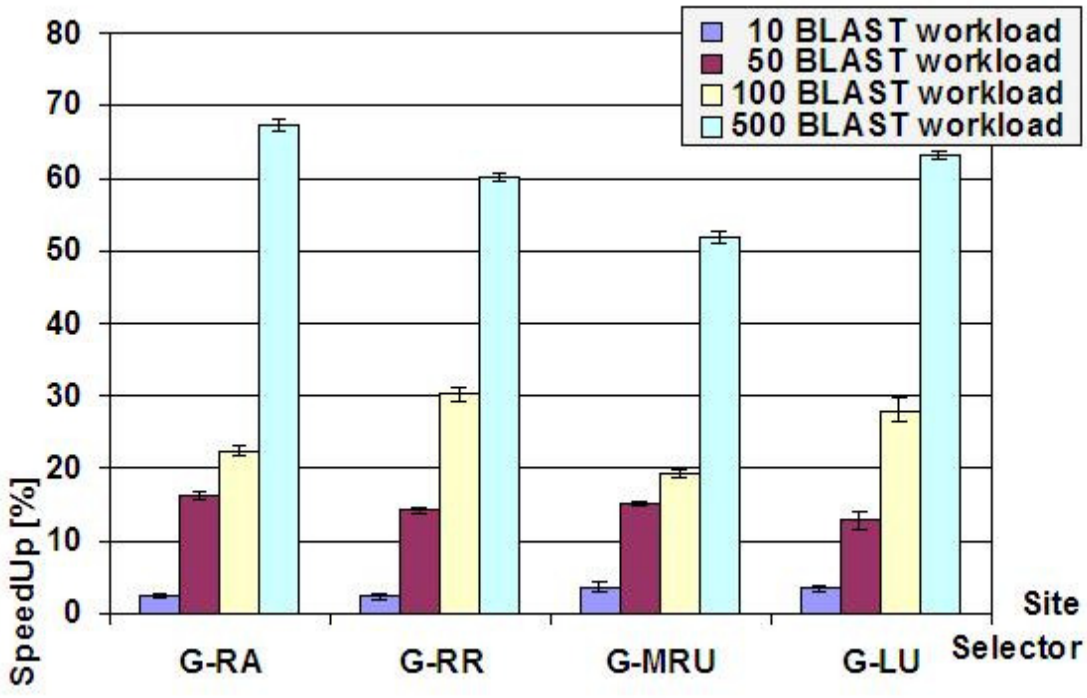


Figure 6.11. Speedup Comparisons among Workloads

Failure Analysis: In this section, I discuss failure results for the version of OSG/Grid3 used for the small and medium workloads (presented in Table 6.13). As can be observed, most of the errors are due to various transient errors in the infrastructure (RLS database, GridFTP servers) or due to the more than 20 minute queue times. OSG/Grid3 performance in executing BLAST workloads increased between the large workload experiments and the small and medium workload experiments: the average **EPJ** of 2.28 for the first set of workloads decreased to 1.3 in this second set of experiments. The largest improvement was achieved for the environment setup (**RUE**). Other errors increased. For example, RLS errors increased due to new bugs in the newer releases.

Table 6.13. Error Percentages of 28k BLAST jobs submitted as small and medium workloads (Percentages are computed as the ratio of current errors to the total number of errors)

Error Description	EPJ	Workload failure (#, %)	Job failure (#, %)
Remote Local Scheduling Timeout	0.56 [-]	5 (3)	15810 (42)
Remote Application Execution Failure	0.21 [-]	0 (0)	5820 (15)
Remote Transient Authentication Error	0.11 [+]	0 (0)	3152 (8)
Transient Database Failure (RLS)	0.19 [+]	0 (0)	5416 (14)
Remote Unset Environment	0.0 [-]	0 (0)	1 (0)
Remote Transient GridFTP Failure	0.18 [+]	0 (0)	5202 (14)
Remote GRAM Environment Error	0.06 [+]	0 (0)	1769 (4)
Others	0.0 [-]	0 (0)	39 (0)
Totals	1.3 [-]	5 (3)	37209 (100)

6.5. Conclusions

Running workloads in Grid environments is often a challenging problem due the scale of the environment and to the resource participation that is based on various sharing strategies. In addition, a resource may be taken down during job execution, be improperly setup, or just fail in job execution. Such elements must be taken in account when targeting a grid environment.

In this chapter, I showed the results that can be achieved for running workloads under an uSLA-enabled scheduling infrastructure and explored some of the issues that occur in practice. During these experiments I faced various problems as described above, as well as quantified what performance a Grid user should expect. In addition, I observed for my brokering solution that for medium workloads, **G-RA** performs best with a 90% confidence interval, while **G-LU** performs best for smaller workloads. I also note that **G-MRU** performed worst for all tested workloads.

CHAPTER SEVEN

CONCLUSIONS AND FUTURE WORK

This dissertation focuses on controlled resource sharing in distributed environments, more specifically in Grid environments. The characteristics of this target environment, a large number of participants, namely resources and resource providers on one side and users and user communities on the other part, require solutions that are both scalable and decentralized. In addition, there are situations when privacy and reliability are important. I designed a solution for controlled resource sharing starting from previous work in the networking domain and extended it for aggregated and complex resources that participate in large and dynamic environments.

7.1. Lessons

The most important lesson to draw from this work is that controlled resource sharing is difficult in practice, due to the number and the complexity of participants, their local preferences and software, but possible. Also, the four uSLA semantics proposed in Chapter 3 proved sufficient for the OSG/Grid3 resource sharing scenario plus two other generic scenarios [31, 67].

The experimental results of Chapter 6 are one of the weak points of this dissertation. Even though the number of experiments presented is large compared to other similar work [80, 134], additional applications and runs would have made the analysis much stronger. Unfortunately, OSG/Grid3 availability and its permanent evolution made it impossible to obtain of more detailed results.

As a final note, while GangSim simulator and GRUBER broker [42, 43] could be used or deployed for a large set of similar scenarios and environments, the generality of the uSLA idea make appealing for application to new scenarios that require controlled resource sharing. For example, I have explored the possibility of enabling a specialized Grid scheduler developed for DAS-2 [135], the Dutch Grid system, with support for uSLA-based resource provisioning in multi-virtual domain environments [48, 136].

In conclusion, I believe that uSLA-based resource sharing provides a strong starting point for building environments in which resources are shared under owner preferences. This approach can be augmented to allow for additional mechanisms for finer resource control [22, 39].

7.2. Future Research Directions

As future work, I am interested in expanding the set of possible uSLA semantics, improving GangSim, perform future validation studies on GRUBER, and exploring other mechanisms for controlled resource sharing in large cooperative environments.

While the uSLAs proposed in this thesis are comprehensive, I expect that in the future new semantics will be required. An immediate scenario that requires such

enhancements is represented by the current effort pursued by various Grids in Europe to interoperate. I am analyzing decay-based and utilization-based uSLAs [48].

I plan to validate GangSim against real Grids. Preliminary simulator validation studies show inconclusive results, in part, because Grids introduce latencies and failures that are not captured by GangSim. Second, I plan to extend GangSim with a language-based uSLA specification and support for scheduling studies for virtual domain resource sharing [48]. And third, I intend to study whether running GangSim in a distributed mode in which several simulator instances run on different hosts, with sites and computational load distributed appropriately, can support larger scenarios than the ones presented in Chapter 4.

I would like to deploy the DI-GRUBER framework in a real Grid. Also, I am interested in large Grid environments because of the unexpected challenges that such environment might present in practice.

I plan to compare economy-based models for resource sharing with this work. For example, I am interested in pursuing lottery scheduling where currency and ticket abstractions serve as a means of flexible allocations [16]. Such an approach will not replace the uSLA-based scheduling, but can support the leveraging of some parts pf the current proposed infrastructure (i.e., verifiers' elimination), and finer granularity for resource sharing [39].

REFERENCES

1. Oetiker, T., *Tobi Oetiker's MRTG - The Multi Router Traffic Grapher*. 2004.

2. Oetiker, T., *RRDtool: A system to store and display time series*. 2005.

3. Foster, I., *The Grid: A New Infrastructure for 21st Century Science.* Physics Today. **55**(2): p. 42-47.

4. Foster, I., C. Kesselman, and S. Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations.* International Journal of Supercomputer Applications, 2001. **15**(3): p. 200-222.

5. Crowcroft, J., et al., *Peer-to-Peer Technologies*, in *The Grid: Blueprint for a New Computing Infrastructure (2nd Edition)*. 2004, Morgan-Kaufmann.

6. Ripeanu, M. and I. Foster. *Mapping the Gnutella Network: Macroscopic Properties of Large-Scale Peer-to-Peer Systems*. in *1st International Workshop on Peer-to-Peer Systems*. 2002.

7. Ripeanu, M. and I. Foster. *A Decentralized, Adaptive, Replica Location Service*. in *11th IEEE International Symposium on High Performance Distributed Computing*. 2002. Edinburgh, Scotland: IEEE Computer Society Press.

8. Rowstron, A. and P. Druschel. *Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems*. in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*. 2001.

9. Rowstron, A.I.T. and P. Druschel. *Storage Management and Caching in PAST, a Large-Scale, Persistent P2P Storage Utility*. in *Symposium on Operating Systems Principles*. 2001.

10. Stonebraker, M., et al., *Mariposa: A Wide-Area Distributed Database System.* VLDB Journal, 1996. **5**(1): p. 48-63.

11. Ghemawat, S., H. Gobioff, and S.T. Leung. *The Google File System*. in *SOSP*. 2003.

12. Foster, I. *Grid Computing*. in *Advanced Computing and Analysis Techniques in Physics Research (ACAT)*. 2000: AIP Conference Proceedings.

13. Foster, I. and others. *The Grid2003 Production Grid: Principles and Practice*. in *IEEE International Symposium on High Performance Distributed Computing*. 2004: IEEE Computer Science Press.

14. Pearlman, L., et al. *A Community Authorization Service for Group Collaboration*. in *IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*. 2002.

15. Globus Project, *Community Authorization Service (CAS).* 2002.

16. Zhao, T. and V. Karamcheti, *Expressing and Enforcing Distributed Resource Agreements*. 2000, Department of Computer Science, Courant Institute of Mathematical Sciences: New York.

17. Raman, R., *Matchmaking Frameworks for Distributed Resource Management*. 2000, University of Wisconsin.

18. Foster, I., et al., *End-to-End Quality of Service for High-end Applications.* Computer Communications, 2004. **27**(14): p. 1375-1388.

19. Czajkowski, K., et al. *SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems*. in *8th Workshop on Job Scheduling Strategies for Parallel Processing*. 2002. Edinburgh, Scotland.

20. Thain, D., T. Tannenbaum, and M. Livny, *Condor and the Grid*, in *Grid Computing: Making The Global Infrastructure a Reality*, A. Hey, Editor. 2003, John Wiley.

21. Ludwig, H., A. Dan, and B. Kearney. *Cremona: An Architecture and Library for Creation and Monitoring WS-Agreements*. in *ACM International Conference on Service Oriented Computing (ICSOC'04)*. 2004. New York.

22. Dan, A., et al., *Web Services on Demand: WSLA-driven automated management*, S. Journal, Editor. 2004, IBM. p. 136.

23. Dan, A., C. Dumitrescu, and M. Ripeanu. *Connecting Client Objectives with Resource Capabilities: An Essential Component for Grid Service Management Infrastructures*. in *ACM International Conference on Service Oriented Computing (ICSOC'04)*. 2004. New York.

24. Keller, A. and H. Ludwig, *The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services.* Journal of Network and Systems Management, 2003. **11**(1).

25. IBM, *IBM, Globus announce open grid services.* 2002.

26. Verma, D.C., *Policy Based Networking, Architecture and Algorithm*. November 2000: New Riders Publishing.

27. Dumitrescu, C. *INTCTD: A Peer-to-Peer Approach for Intrusion Detection*. in *Cluster Computing and Grids (CCGrid'06)*. 2006. Singapore.

28. Verma, D.C., *Simplifying Network Administration using Policy based Management*. 2004, IBM.

29. Lamanna, D., J. Skene, and W. Emmerich. *SLang: A Language for Defining Service Level Agreements*. in *the 9th IEEE Workshop on Future Trends in Distributed Computing Systems*. 2003: IEEE Computer Society Press.

30. Chervenak, A., et al., *The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Data Sets.* J. Network and Computer Applications, 2001(23): p. 187-200.

31. Foster, I. and e. al. *The Grid2003 Production Grid: Principles and Practice*. in *13th International Symposium on High Performance Distributed Computing (HPDC)*. 2004.

32. *LHC Computing Project*. 2004.

33. Dumitrescu, C., M. Wilde, and I. Foster. *A Model for Usage Policy-based Resource Allocation in Grids*. in *Proceedings of the Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05)*. 2005. Stockholm, Sweden: IEEE Computer Society.

34. Chervenak, A., E. Deelman, P. Kunszt, E. Laure, A. Shoshani, H. Stockinger, K. Stockinger, D. Thain,, *Data Grid Architecture (Draft)*. 2002.

35. Foster, I., *Data Grids*, in *Databasing the Brain*. 2005, Wiley.

36. Iosup, A., et al. *Synthetic Grid Workloads with Ibis, KOALA, and GrenchMark*. in *CoreGRID Integrated Research in Grid Computing*. 27-30 November 2005. Pisa, Italy.

37. Mohamed, H.H. and D.H.J. Epema. *Experiences with the KOALA Co-Allocating Scheduler in Multiclusters*. in *5th IEEE/ACM Int'l Symp. on Cluster Computing and the GRID (CCGrid2005)*. May 2005. Cardiff, UK.

38. Dumitrescu, C. and I. Foster. *Usage Policy-based CPU Sharing in Virtual Organizations*. in *5th International Workshop on Grid Computing*. 2004. Pittsburgh, PA.

39. Gimpel, H., et al. *PANDA: Specifying Policies for Automated Negotiations of Service Contracts*. in *The 1st International Conference on Service Oriented Computing*. 2003. Trento, Italy.

40. Dumitrescu, C., I. Raicu, and I. Foster. *Experiences in Running Workloads over Grid3*. 2005. Beijing, China: GCC 2005.

41. Mueller, E.T., J.D. Moore, and G.J. Popek. *A Nested Transaction Mechanism for LOCUS*. in *The Ninth ACM Symposium on Operating System Principles (SOSP)*. 1983. Bretton Woods, New Hampshire.

42. Dumitrescu, C. and I. Foster. *GangSim: A Simulator for Grid Scheduling Studies*. in *Cluster Computing and Grid*. 2005. Cardiff, UK.

43. Dumitrescu, C. and I. Foster. *GRUBER: A Grid Resource SLA Broker*. in *Euro-Par*. 2005. Portugal.

44. Quétier, B. and F. Cappello. *A survey of Grid research tools: simulators, emulators and real life platforms*. in *IMACS'2005 - the 17th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*. 2005. Paris, France.

45. Dumitrescu, C., I. Raicu, and I. Foster. *DI-GRUBER: A Distributed Approach for Grid Resource Brokering*. in *SC'05*. 2005. Seattle.

46. Chun B., D.C., T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, *PlanetLab: An Overlay Testbed for Broad-Coverage Services.* ACM Computer Communications Review, 3, July 2003. **33**(3).

47. Foster, I., et al., *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration (extended version of Grid Services for Distributed System Integration)*. 2002, Open Grid Service Infrastructure WG, Global Grid Forum.

48. Dumitrescu, C., et al., *Virtual Domain Sharing in e-Science based on Usage Service Level Agreements*. 2006: Delft.

49. Chervenak, A., et al., *The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Data Sets.* J. Network and Computer Applications, 2000. **23**(3): p. 187-200.

50. Welch, V., et al., *X.509 Proxy Certificates for Dynamic Delegation*. 2004, Globus Alliance.

51. ITU-T, *X.509 (03/00): Information Technology - Open Systems Interconnection - The Directory: Authentication Framework*. 2002, ITU.

52. Thompson, M., et al., *Certificate-based Access Control for Widely Distributed Resources*, in *Proc. 8th Usenix Security Symposium*. 1999.

53. Allcock, W., *GridFTP: Protocol Extensions to FTP for the Grid*. 2003, Global Grid Forum.

54. Foster, I. and C. Kesselman, *The Globus Project: A Status Report*, in *Proc. Heterogeneous Computing Workshop*. 1998, IEEE Press. p. 4-18.

55. AL-ALI, R., et al., *G-QOSM: Grid service discovery using QOS properties.* Computing and informatics (Comput. inform.) ISSN 1335-9150, 2002. **21**(4): p. 363-382.

56. Czajkowski, K., I. Foster, and C. Kesselman, *Resource and Service Management*, in *The Grid: Blueprint for a New Computing Infrastructure (2nd Edition)*. 2004.

57. Czajkowski, K., et al., *Grid Service Level Agreements*, in *Grid Resource Management*. 2003, Kluwer.

58. Czajkowski, K., et al., *Agreement-Based Grid Service Management (WS-Agreement)*. 2003, Global Grid Forum.

59. In, J. and P. Avery. *Policy Based Scheduling for Simple Quality of Service in Grid Computing*. in *International Parallel & Distributed Processing Symposium (IPDPS)*. April '04. Santa Fe, New Mexico.

60. Buyya, R., *GridBus: A Economy-based Grid Resource Broker*. 2004, The University of Melbourne: Melbourne.

61. Xia, H., et al. *The microgrid: Using emulation to predict application performance in diverse grid network environments*. in *Workshop on Challenges*

of Large Applications in Distributed Environments (CLADE'04). June 2004. Honolulu, Hawaii.

62. Casanova, H. *SimGrid: a toolkit for the simulation of application scheduling*. in *IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*. May 2001. Brisbane, Australia.

63. Buyya, R. and M. Murshed. *Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing*. in *The Journal of Concurrency and Computation: Practice and Experience (CCPE)*. 2002.

64. Ranganathan, K. and I. Foster, *Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids.* Journal of Grid Computing, 2003. **1**(1).

65. Legrand, I.C., et al. *MonALISA: A Distributed Monitoring Service Architecture*. in *Computing in High Energy Physics*. 2003. La Jolla, CA.

66. Lupu, E., *A Role-based Framework for Distributed Systems Management*, in *Department of Computing*. 1998, University of London: London.

67. Dumitrescu, C., *Policy Research for iVDGL*. 2004, The University of Chicago / NSF Review: Chicago, IL.

68. Foster, I., et al. *The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration*. in *Conference on Innovative Data Systems Research*. 2003.

69. Global Grid Forum, *Scheduling and Resource Management Area*. 2002.

70. Lumb, I. and C. Smith, *Scheduling Attributes and Platform LSF*, J. Weglarz, Editor. 2003, Kluwer Academic Publishers.

71. Legrand, A., L. Marchal, and H. Casanova. *Scheduling Distributed Applications: The SimGrid Simulation Framework*. in *3rd IEEE International Symposium on Cluster Computing and the Grid*. 2003. Tokyo, Japan.

72. Henry, G.J., *A Fair Share Scheduler.* AT&T Bell Laboratory Technical Journal, October 1984. **3**(8).

73. Schroeder, B. and M. Harchol-Balter. *Evaluation of Task Assignment Policies for Super Computing Servers: The Case for Load Unbalancing and Fairness*. in *Cluster Computing*. April 2004.

74. IBM, Microsoft, and VeriSign, *Web Services Security Language (WS-Security)*. 2002.

75. Ranganathan, K. and I. Foster. *Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications*. in *11th IEEE International Symposium on High Performance Distributed Computing*. 2002. Edinburgh, Scotland: IEEE Computer Society Press.

76. Mambelli, M., *Capone and VDS*. 2005, The University of Chicago and Argonne National Laboratory: Chicago.

77. Annis, J., S. Kent, and A. Szalay, *The SDSS-GriPhyN Challenge Problems: Cluster Finding, Correlation Functions and Weak Lensing*. 2001, FermiLab.

78. Sulakhe, D., *Genome Analysis Research Environment*. 2005, Argonne National Laboratory: Chicago.

79. Foster, I., et al. *Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation*. in *14th Intl. Conf. on Scientific and Statistical Database Management*. 2002. Edinburgh, Scotland.

80. Iosup, A., et al. *How are Real Grids Used? The Analysis of Four Grid Traces and Its Implications*. in *Grid Conference*. 2006. Barcelona, Spain: IEEE Communication Society.

81. *Open Science Grid (OSG)*. 2004.

82. *DOE Science Grid PKI Certificate Policy and Certification Practice Statement*. 2002.

83. Dumitrescu, C., *ARESRAN: A WSRF-based Resource Reservation Service for Grid Service*. 2005.

84. Dan, A., et al., *A Layered Framework for Connecting Client Objectives and Resource Capabilities.* International Journal of Cooperative Information Systems, 2006(2006).

85. M. Humphrey, G.W., K. Jackson, J. Boverhof, M. Rodriguez, Joe Bester, J. Gawor, S. Lang, I. Foster, S. Meder, S. Pickles, and M. McKeown. *State and Events for Web Services: A Comparison of Five WS-Resource Framework and WS-Notification Implementations*. in *4th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*. 24-27 July 2005. Research Triangle Park, NC.

86. Foster, I. and C. Kesselman, eds. *The Grid: Blueprint for a New Computing Infrastructure (2nd Edition)*. 2004, Morgan Kaufmann.

87. Dumitrescu, C., M. Wilde, and I. Foster, *Usage Policies at the Site Level In Grid*. 2004, iVDGL/GriPhyN Project: Chicago.

88. Massie, M., B. Chun, and D. Culler. *The Ganglia Distributed Monitoring: Design, Implementation, and Experience*. in *Parallel Computing*. May 2004.

89. Wisconsin, O.-U.o., *UWMadisonCMS Open Science Grid Site Policy Page*. 2006, University of Wisconsin, Madison.

90. FNAL, O.-. *FNAL_GPFARM Site Policy for OSG*. 2006, FNAL.

91. USCMS, O.-V., *USCMS - OSG Policy Pages*. 2006, USCMS.

92. Keahey, K., T. Araki, and P. Lane. *Agreement-Based Interactions for Experimental Science*. in *Europar*. 2004.

93. Wolf, L.C. and R. Steinmetz, *Concepts for Reservation in Advance.* Kluwer Journal on Multimedia Tools and Applications, May 1997. **4**(3).

94. Team, M., *Maui Scheduler*, Center for HPC Cluster Resource Management and Scheduling.

95. Perelmutov, L., J. Bakken, and D. Petravick. *Storage Resource Manager*. in *CHEP04*. 2004. Interlaken, Switzerland.

96. Constandache, I., *Policy Based Dynamic Negotiation for Grid Services Authorization*, in *L3S Research Center*. 2005, University of Hannover: Hannover, Germany. p. 13.

97. Freeman, T., R. Ananthakrishnan, *Authorization processing for Globus Toolkit Java Web services*. October 2005, IBM: New York.

98. Deelman, E., et al. *GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists*. in *11th IEEE International Symposium on High Performance Distributed Computing*. 2002. Edinburgh, Scotland.

99. Deelman, E., et al. *Pegasus : Mapping Scientific Workflows onto the Grid*. in *Across Grids Conference 2004*. 2004. Nicosia, Cyprus.

100. Vöckler, J.-S., M. Wilde, and I. Foster, *The GriPhyN Virtual Data System*. 2002.

101. Litzkow, M., M. Livny, and M. Mutka, *Condor - A Hunter of Idle Workstations*, in *Proc. 8th Intl Conf. on Distributed Computing Systems*. 1988. p. 104-111.

102. *PBS website*. 2004.

103. *LSF 2.2 User's Guide.* February 1996.

104. Livny, M. and e. al, *Condor*.

105. Zhou, S. *LSF: Load Sharing in Large-Scale Heterogeneous Distributed Systems*. in *Workshop on Cluster Computing*. 1992.

106. De Jongh, J.F.C.M., *Share Scheduling in Distributed Systems*, in *Delft Technical University*. 2002.

107. Dumitrescu, C., et al., *User-Transparent Scheduling of Higher Order Components in Grid Environments*. 2006, CoreGrid Network of Excellence: Delft/Muenster.

108. Lee, R. and J. Skovronski, *Grid Simulators: A Survey Of Various Simulators And An Outlook On The Future Of Grid Simulation*. 2005, Cornell University: New York.

109. Foster, I., et al., *Grid Services for Distributed Systems Integration.* IEEE Computer, 2002. **35**(6): p. 37-46.

110. Constantinescu, Z., P. P., and P. A. *Q2ADPZ: An Open system for distributed computing*. in *The fourth Nordic EurOpen/USENIX Conference, NordU2002*. 2002. Norway.

111. Henderson, R. and D. Tweten, *Portable Batch System: External Reference Specification.* 1996.

112. Litzkow, M.J., M. Livny, and M.W. Mutka, *Condor - A Hunter of Idle Workstations*, in *8th International Conference on Distributed Computing Systems*. 1988. p. 104-111.

113. Foster, I., C. Kesselman, and S. Tuecke, *The Globus Toolkit and Grid Architecture*. 2001, In preparation.

114. Foster, I., et al., *Modeling Stateful Resources with Web Services*. 2004: www.globus.org/wsrf.

115. Foster, I., C. Kesselman, and S. Tuecke, *The Open Grid Services Architecture*, in *The Grid: Blueprint for a New Computing Infrastructure (2nd Edition)*. 2004, Morgan Kaufmann.

116. Czajkowski, K., et al. *Grid Information Services for Distributed Resource Sharing*. in *10th IEEE International Symposium on High Performance Distributed Computing*. 2001: IEEE Computer Society Press.

117. Dumitrescu, C., et al. *DiPerF: Automated DIstributed PERformance testing Framework*. in *5th International Workshop in Grid Computing*. 2004.

118. Dumitrescu, C. *Problems for Resource Brokering in Large and Dynamic Grid Environments*. in *European Conference on Distributed Systems (Euro-Par)*. 2006. Dresden, Germany: LNCS.

119. Team, V., *VDT Release Documentation*. 2006, The University of Wisconsin: Wisconsin.

120. Team, V., *GriPhyN - The Virtual Data System*. 2006, The University of Chicago / GriPhyN Project: Chicago.

121. Dumitrescu, C., I. Raicu, and I. Foster, *Usage SLA-based Scheduling in Grids.* Journal of Concurrency and Computation: Practice and Experience (GCC05-Special Issue), 2006.

122. *The GriPhyN/PPDG Virtual Data Toolkit (VDT)*. 2004.

123. Team, T.G., *The Globus Toolkit*. 2006, Globus Consortium: Chicago.

124. Frey, J., et al., *Condor-G: A Computation Management Agent for Multi-Institutional Grids.* Cluster Computing, 2002. **5**(3): p. 237-246.

125. Avery, P. and I. Foster, *The GriPhyN Project: Towards Petascale Virtual Data Grids*. 2001.

126. *Maui Scheduler*: http://supercluster.org/maui.

127. Condor Project, *Personal Condor and Globus Glide-In*. 2002.

128. Team, O.-P., *A Batching Queuing System, Software Project*, Altair Grid Technologies, LLC.

129. Cluster Resources, I., *Maui - PBS Integration Guide*. 2006, Cluster Resources, Inc.

130. Foster, I. and C. Kesselman, *Globus: A Toolkit-Based Grid Architecture*, in *citeGridBook*. p. 259--278.

131. Collaboration, T.G., *The GriPhyN Project*. 2000, www.griphyn.org.

132. Livny, M., *DAGMAN reference.*

133. Chervenak, A., et al., *A Scalable Replica Location Service*. 2001, Work in progress.

134. Iosup, A. and D. Epema. *GrenchMark: A Framework for Analyzing, Testing, and Comparing Grids*. in *the 6th IEEE/ACM Int'l Symposium on Cluster Computing and the Grid (CCGrid'06)*. 2006. Singapore: IEEE Computer Science Society.

135. Backbone, D.U., *DAS-2 (Distributed ASCI Supercomputer)*. 2006: Amsterdam.

136. Dumitrescu, C., et al. *Reusable Cost-based Scheduling for Higher-Order Components over the Grid*. in *The 2nd e-Science Conference 2006*. 2006. Amsterdam, the Netherlands.

137. Hill N., Watson A., Stone P., Friedman J., EMEA High Performance On Demand Solutions Team, Case study: SOA-based On Demand Service Delivery Solution at Star Technology Services Ltd., March 13, 2006 Online:
ftp://ftp.software.ibm.com/software/dw/wes/hipods/Star-WhitePaper3-13-2006.pdf

138. IBM - High Performance on Demand Solutions, *On Demand Service Delivery forSOA Environments,* Online: ftp://ftp.software.ibm.com/software/dw/wes/hipods/SOA_summit_wp9Mayf.pdf, Last accessed: March 8, 2007.

139. R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef, "Performance management for cluster based web services," in Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management, (Colorado Springs, CO), March 2003.