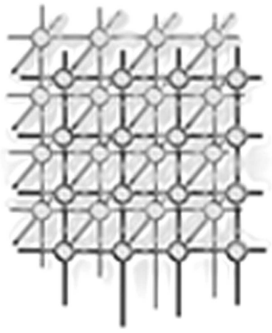
Submitted to CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
October 2006

Scalable Grid-wide capacity allocation with the SweGrid Accounting System (SGAS)



Peter Gardfjäll[‡], Erik Elmroth[‡], Lennart Johnsson[§], Olle Mulmo[§], and Thomas Sandholm[§]

[‡] *Dept. of Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden*
[§] *School of Computer Science and Communication, Royal Institute of Technology, SE-100 44 Stockholm, Sweden*

SUMMARY

The SweGrid Accounting System (SGAS) allocates capacity in collaborative Grid environments by coordinating enforcement of Grid-wide usage limits to offer usage guarantees and prevent overuse. SGAS employs a credit-based allocation model where Grid capacity is granted to projects via Grid-wide quota allowances that can be spent across the Grid resources. The resources collectively enforce these allowances in a soft, real-time manner.

SGAS is built on service-oriented principles with a strong focus on interoperability and Web services standards. This article covers the SGAS design and implementation, which besides addressing inherent Grid challenges (scale, security, heterogeneity, decentralization) emphasizes generality and flexibility to produce a customizable system with lightweight integration into different middleware and scheduling system combinations.

We focus the discussion around the component design; a flexible allocation model; middleware integration experiences; scalability improvements via a distributed virtual banking system; and, finally, an extensive set of testbed experiments. The experiments evaluate the performance of SGAS in terms of response times, request throughput, overall system scalability, and its performance impact on the Globus Toolkit 4 job submission software. We conclude that, for all practical purposes, the overhead incurred by SGAS on job submissions will not be a limiting factor for the job handling capacity of the job submission software.

KEY WORDS: Grid accounting; Grid capacity allocation; quota enforcement; service virtualization; service-oriented architecture (SOA); Web services; Globus Toolkit;

[‡]E-mail: peterg@cs.umu.se, elmroth@cs.umu.se
[§]E-mail: johnsson@tlc2.uh.edu, mulmo@pdc.kth.se, sandholm@pdc.kth.se
Contract/grant sponsor: Swedish Research Council (VR); contract/grant number: 343-2003-953, 621-2005-3667

1. INTRODUCTION

As an enabler of large-scale resource sharing, Grid technology promises access to unprecedented amounts of computing capacity by integrating pools of computational resources across organizational boundaries, presenting them to users as a single virtual (Grid) system. An important objective for the Virtual Organizations (VOs) [19] that result from these sharing arrangements is to make efficient use of the provisioned resource capacity to maintain a high degree of overall system utilization and satisfy individual project's service needs.

Lacking capacity allocation mechanisms that operate across the Grid, the capacity of most Grid systems to date have been completely unregulated, essentially making the Grid a "source of free CPU cycles" for authorized users. When unrestricted access is admitted to a shared resource, the pursuit of the individual good eventually causes over-exploitation and degradation of the common resource – a phenomenon often referred to as the "tragedy of the commons" [29]. Apart from preventing such overuse, it is important to be able to offer differentiated usage guarantees to accommodate the differing needs and importance of projects. We address both of these issues with the SweGrid Accounting System (SGAS) [45], a Grid accounting system that records and coordinates usage across the Grid to enforce Grid-wide capacity allotments.

Enforcing usage limits for cluster users has been common practice at HPC centers for a long time. Although extending this concept to Grids may seem simple from a conceptual perspective (after all a Grid can be regarded as a single, although distributed, computational resource) the inherent challenges of Grid environments (large scale; decentralized management; hardware, platform, data format heterogeneity; cross-organizational security) makes it a difficult problem.

SGAS, which has been built using the Web services primitives of Globus Toolkit (GT) 4 [17], is currently included as a technology preview in GT4. Although SGAS was primarily targeted towards usage logging and cross-site enforcement of resource allocations within SweGrid, SGAS has been designed to allow simple integration into existing middleware and scheduling systems in an attempt to address the inherent Grid heterogeneity. An early prototype, developed using GT3, was presented in [42] and [14]. A revised version of [42] was later published in [43].

The main contributions of this article include a flexible allocation model based on the notion of time-stamped allocations; detailed component descriptions and design rationale; experiences from middleware integration with GT4 and the Advanced Resource Connector (ARC) [47]; improved scalability via a banking system that is virtualized across several servers; and, finally, an extensive performance and scalability evaluation.

The rest of the article is organized as follows: Section 2 describes SGAS and its perceived operation context. The architecture of the SGAS framework is outlined in Section 3 and the design and implementation of the SGAS components is covered in Section 4. The performance of SGAS is analyzed in Section 5 and, finally, we present related work in Section 6 and some concluding remarks in Section 7.

2. A CAPACITY ALLOCATION MECHANISM FOR THE GRID

SGAS provides system-level services for coordinating usage across the Grid by logically dividing the aggregate capacity of the VO-provisioned resources between user groups. In essence, SGAS performs

two independent tasks: management and enforcement of Grid-wide capacity allotments on the one hand, and usage tracking across the heterogeneous set of Grid resources on the other. This results in two basic modes of operation for SGAS – logging mode and allocation enforcement mode (which includes logging). Besides being necessary for allocation enforcement, usage tracking also forms a basis for site and/or VO usage monitoring, audit trails, and the ability to charge Grid usage (whether real or virtual money). Usage monitoring can in turn be used by site or VO management to evaluate return on investment and perform capacity planning.

For capacity allocation, SGAS employs a credit-based model, expressed in terms of Grid-wide quotas that are granted to user groups, representing an entitled share of the Grid capacity. These quotas are spent across the Grid resources by users, who pay for the resources they consume. Allocations are enforced in a decentralized manner by the Grid resources, which, at the time of job submission, may reject a job request from a user that has exceeded its quota. We characterize the SGAS enforcement mechanism as being "real-time", since enforcement is carried out at the time of job submission, and "soft" since the degree of enforcement strictness is subject to stakeholder policies. Under a strict enforcement policy, all quota-exceeding jobs are disallowed. Under a softer enforcement mode a credit limit can be introduced to allow temporary quota exceeding, which gives additional flexibility in spending to account for unpredictable computational needs. Furthermore, resource owners may allow quota-exceeding jobs to improve local utilization at times of light load.

Although we do not exclude use in commercial pay-per-use/utility computing environments, where Grid credits would be traded for real money, we have mainly focused our efforts towards collaborative Grid environments, such as those often found in scientific settings, where academic/research institutes pool their computational resources. In this context we distinguish three main stakeholders that our system serves, as illustrated in Figure 1:

- Resource owners, who contribute resources to VOs.
- Allocation authorities/committees, which divide the aggregate VO capacity between user groups.
- User groups, who consume Grid resources (subject to the restrictions imposed by the allocation authority).

These stakeholders have differing, and to some extent conflicting, goals. The allocation authority wants to make optimal use of VO capacity and to that end grants different-sized Grid capacity allotments to projects on the basis of their computational needs and research contribution. This capacity allocation requires resources to coordinate their enforcement efforts. However, with the decentralized management structure of Grids, such coordination cannot be forced upon resource owners, who always retain ultimate control of their resources. Owners may have goals of their own, a common one being to achieve high utilization. Utilization improvements are usually achieved at the expense of fairness, by relaxing usage limits or allowing out-of-order execution of jobs (e.g. backfilling [30]). This may, however, conflict with user interests, which typically involves QoS guarantees and (strict) fairness (that is, receiving the capacity entitlement).

In summary, there is a conflict between global and local resource control and there is a trade-off between resource utilization and fairness. SGAS, as the mediating capacity allocation mechanism, tries to satisfy the needs of all stakeholders, by offering flexible policy customization for the three parties that the system serves. It strives to allow allocation authorities to divide the aggregate VO capacity between users in a fair manner and coordinate allocation enforcement across the Grid without sacrificing resource owner autonomy.

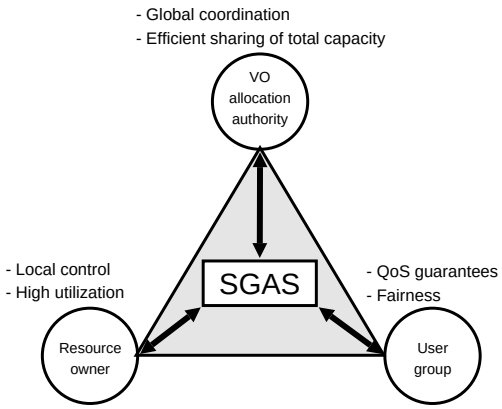


Figure 1. SGAS as the mediator between the parties of the stakeholder triangle.

Examples of environments that fit this model include research Grids such as Swegrid [51] and TeraGrid [52]. The CPU time of the Swegrid computational resource (integrating six 100-CPU high-performance clusters) is controlled by the Swedish National Allocation Committee (SNAC) [48] allocation authority, which grants CPU time on Swegrid to research projects that require substantial access to computing power. These projects are granted portions of the Grid capacity after a peer-reviewed application process. The TeraGrid counterpart of SNAC is the National Resource Allocation Committee (NRAC). However, the model is not limited to research settings. Similar collaborative sharing arrangements are easy to conceive in intra-organizational Grids, spanning several offices within a single company, or between industry partners in company-crossing, inter-organizational Grids.

3. SYSTEM ARCHITECTURE

The loosely coupled and dynamic nature of Grid environments impose additional requirements on Grid software, as compared to software written for more tightly coupled cluster computing systems and traditional (single-organization) distributed computing systems.

Grid systems are typically large-scale systems that integrate widely distributed computational resources that belong to different administrative (and security) domains. Therefore, management in Grids tends to be decentralized, where resource owners retain control over their local resources. The computational resources that are shared by the VO participants can be heterogeneous in terms of hardware, platform, software stacks and data formats. Since VO members constitute a wider and less trusted user community, security becomes more important as compared to tightly coupled systems. These inherent characteristics of Grid environments [50] form the basis for the SGAS design, which addresses the issues of scale, site autonomy, security, heterogeneity and interoperability.

3.1. System Components

Allocation enforcement and usage tracking is carried out by three main SGAS components, which operate together across the Grid sites to provide VO-wide functionality. Two components, the Bank service and the Logging and Usage Tracking Service (LUTS), are remote Web services while the third component, the Job Account Reservation Manager (JARM), is the resource integrator.

The Bank manages a set of accounts, which correspond to the VO project allocations, and maintains a consistent view of the resources consumed by each project in order to coordinate quota enforcement across the Grid sites.

The LUTS tracks usage across the Grid, by allowing Grid sites to publish detailed information about completed jobs in the uniform, XML-based Usage Record format [53]. These usage records can be queried by means of the XPath query language [8].

JARM integrates Grid resources with SGAS, serving as the single-point-of-integration with the underlying middleware and the Grid workload manager or resource allocator. Each incoming job request on the resource is intercepted by JARM, which makes a callout to the bank on behalf of the user (with the user's delegated proxy) to acquire a reservation on a portion of the project's allocation prior to servicing the request, thereby enforcing project quotas and preventing over-use. Upon job completion, the account is charged for the consumed resources and a usage record is logged in the LUTS. These interactions are illustrated in Figure 2. A more detailed description of these components is given in Section 4.

Note that the Bank and LUTS are remote Web services and, as such, completely independent of the middleware solution in the targeted Grid environment. JARM, on the other hand, which has been designed for simple integration with the underlying environment, provides plugin-points for the adapter code and middleware-specific behavior that is required to integrate JARM with new Grid software stacks. To date, JARM has been integrated with two middlewares: GT4 and ARC, both of which may run on top of different scheduling systems (as shown in the figure) with different usage data formats.

3.2. A Service-oriented Approach

The key enabler of long-term Grid computing success is interoperability, which is required to allow seamless operation and virtualization across administrative domains and a heterogeneous resource base. In distributed environments, interoperability requires standard protocols.

We have adopted the service-oriented system principles proposed by the Global Grid Forum (GGF) [21] through the Open Grid Service Architecture (OGSA) [38]. OGSA describes a core Grid computing architecture defined in terms of service interfaces that provide Grid access through Web services, or equivalently, deliver the Grid as service.

In particular, SGAS has been built around the Web Services Resource Framework (WSRF) [22] family of specifications. WSRF complements "vanilla" Web services with support for management of application state and mechanisms for handling frequently recurring tasks in stateful interaction contexts, such as state introspection and soft-state handling of system resources. Throughout this article, we will refer to the standardization efforts that we have employed.

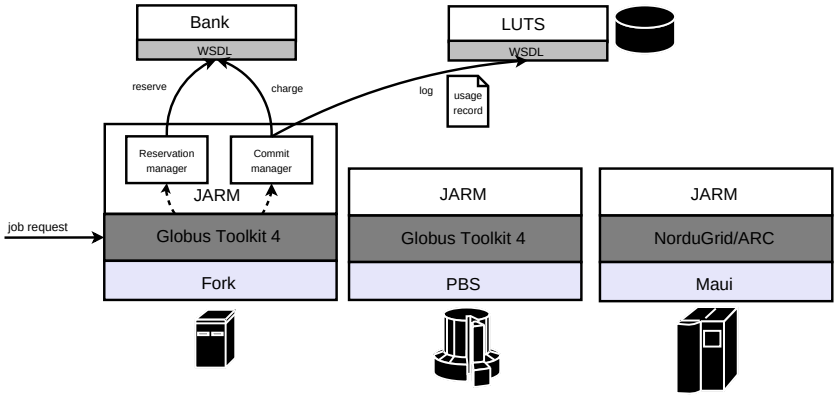


Figure 2. SGAS system overview.

4. DESIGN AND IMPLEMENTATION

Apart from addressing the requirements imposed by large-scale Grid environments (scalability, site autonomy, security, heterogeneity, fault-tolerance), the SGAS design rationale is characterized by focusing on:

- Flexibility. SGAS is intended to be applicable in a wide variety of environments. Therefore, SGAS has been developed independently of any particular platform, middleware, and scheduling system. Instead, SGAS provides plugin-points for environment-specific adapter code and supports policy customization and extensibility along several dimensions. Moreover, SGAS is not limited in terms of its resource usage scope, but is capable of accounting for different (unforeseen) types of resource usage and can be extended to incorporate different pricing models for resource usage rates.
- Non-intrusiveness. SGAS neither replaces existing local accounting and scheduling systems, nor does it mandate specific hardware/software combinations or usage data formats.
- Simple deployment. SGAS should be simple to deploy into the existing infrastructure, without the need to rewrite underlying software. SGAS provides a single resource integration-point via call-outs or plug-ins.
- Transparency. The system only places a marginal additional burden on end-users. In most cases users can remain unaware of that the Grid is “accounting-enabled”.

The SGAS software package, which is available under an open-source license, can be downloaded from the SGAS web site [45]. SGAS has been developed using GT4 (Java WS Core) [17], a Web service development framework that adds WSRF primitives to the Axis Web service engine [2]. These primitives can be combined to build OGSA-compliant Web services. The implementation is entirely Java-based, which results in portable code that runs on any platform with Java support.

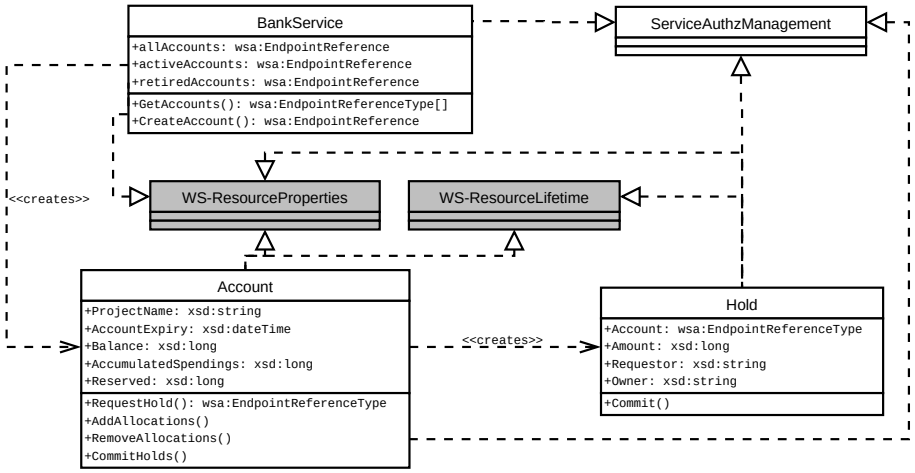


Figure 3. Bank services.

4.1. Bank

The bank is a key component in SGAS that manages project allocations and coordinates allocation enforcement across the Grid sites. These credit allocations are hosted by the bank as a set of independently administered accounts, whose quotas count towards the Grid as a whole (not to individual machines). The allocation credits are unit-less, and can hence be used to charge for arbitrary resource usage (only CPU-time is currently charged). In fact, the bank is completely unaware of the meaning of the credits. The resources, however, need to translate the resources consumed by the job into credits before charging the account. This is done by applying a transformation (according to some pricing scheme) to the job usage.

4.1.1. Service Interfaces

The bank is composed of three WSRF-compliant Web services (Bank, Account, Hold), whose relationships are shown in Figure 3. In this diagram, shaded components are WSRF-defined components, resource properties are shown as attributes, and operations are shown as methods.

New accounts are created through the Bank service, which implements the WS-Resource factory pattern, as described in [18]. Since a VO participant may belong to several VO projects, a user may be a member of several bank accounts. Therefore, the Bank service provides account searches based on the identity of the caller. These searches return endpoint references (EPRs) [27] for all user accounts and can be used by resources to find a chargeable account for a job request that lacks an explicit account reference.

The creation of an account produces an Account WS-Resource representing the project allocation. The Account service interface contains operations for managing the project allocation (adding and removing) and setting up fine-grained access permissions for the account, for instance, determining who may charge the account, update allocations and modify access rights. This is built around the same general authorization framework (described in Section 4.4) upon which all SGAS services rely (the ServiceAuthzManagement interface). Account members may reserve a portion of the account allocation. This is performed at job submission time by the Grid resource, which makes a reservation on behalf of the user (using delegated credentials) to guarantee that the job can be charged when finished. This is referred to as requesting a hold on the allocation.

A successful hold request results in the creation of a Hold WS-Resource. A Hold provides a commit operation, which may only be invoked by the hold owner (the resource that acquired the hold) to charge for the resources consumed by the job. This results in a withdrawal from the hold account and the addition of a transaction entry to the account transaction log. The debited amount may differ from the reserved amount, since they represent actual and approximated usage, respectively. A resource may use a slight overbooking strategy when requesting the hold. The transaction entry contains an EPR to correlate the transaction with its corresponding logging service usage record.

The Account service provides a batch-mode commit operation that charges several holds in a single invocation. This reduces bank traffic by allowing resources to defer job charging and spool commits to perform them periodically in batches. The authorization framework also allows overdraft policies to be established, allowing temporary negative account balance and some additional flexibility in quota spending. The account refuses any reservation attempts that violates the credit limit. However, the bank decision can be overruled by JARM, which depending on local site policies, may still allow the job to execute, for example, to improve local utilization.

All services publish their state via the get/query operations provided by the WS-ResourceProperties specification [25]. For example, the account transaction log can be queried by account members through these operations. Holds are created with a (renewable) lifetime, using the soft-state approach of WS-ResourceLifetime [49], as a safety measure to assure that a reservation is released even if the resource fails after being granted a hold.

As we shall see in Section 4.5, the bank is not confined to a single site. It can be virtualized across several distributed servers to balance load and scale up with larger Grid environments. Furthermore, the distribution reduces the risk of total bank “outage”.

4.1.2. The SGAS Allocation Model

SGAS employs an allocation model where users pay for resource usage in (virtual currency) credits granted by an allocation authority. Typically, these allocations represent CPU time entitlements, although they really model time-limited shares of Grid capacity. Given that resources collectively enforce these allocations in a fairly strict manner, users are likely to feel that they are treated fairly if they receive their entitled share of resources. However, the volatile nature of CPU time may cause a subtle problem with the credit-based allocation model, which may lead to situations where the feeling of fairness is replaced by frustration of not being able to spend the entire allocation.

Part of the problem is that CPU time is a non-storable resource (unused CPU cycles are lost). In combination with the credit model, this can lead to situations where there is an imbalance between the actual available capacity and the modelled capacity (represented by the credits in circulation). That is,

inactive projects may save credits that, eventually, will not have any correlation to actual CPU time, since their cycles have already been lost. This may cause contention problems at allocation period borders, when all projects attempt to spend their remaining quota, since the credits-capacity imbalance makes it impossible for the resources to deliver all project allocations. We end up in a situation where projects are unable to spend their quotas.

SGAS partially solves, or at least reduces, this "inflation" problem through the use of time-stamped allocations. An account has an associated set of allocations, each with a limited validity period during which it is chargeable. Hence, unused project allocations eventually perish, thereby mitigating the saved quota problem. This mechanism reduces the credits-capacity imbalance by continuously revoking surplus credits to better match the actual remaining capacity of the allocation period.

The time-stamped allocation model is flexible as it offers fine-grained control and allows allocation strategies along different dimensions to be implemented. First, quota can be distributed over time. Thus, instead of issuing a single allocation for an entire allocation period, the allocation is broken up into smaller allocations with different (perhaps overlapping) validity periods. This helps prevent contention on allocation period ends and encourages projects to spread their workload in time, potentially resulting in better (more even) resource usage. The model also supports customizing "allocation density" over time according to project needs so that a project allocation can be concentrated to a narrow time-frame, e.g. close to a publication deadline, while less quota is spread over the remaining time to allow for shorter testruns and simulation tuning.

Second, the time-stamped allocation model permits validity periods of arbitrary duration. The choice of validity period length represents a trade-off between flexible utilization and close credit-capacity correlation. Although long validity periods allows more flexibility in quota spending (over time) and hence may improve overall utilization, they increase the risk for quota accumulation and period-end contention. Short allocation periods, on the other hand, facilitate closer credit-capacity correlation (reducing accumulation effects) at the expense of flexible utilization, which potentially may lead to inefficient resource usage.

Third, different approaches to capacity planning are conceivable. Yet again facing a trade-off, this time between fairness and utilization, an allocation authority may choose either an under- or over-provisioning approach when issuing resource grants. The under-provisioning approach may lead to poor utilization since not all users may utilize their full share, while contract/quota fulfillment and fairness becomes easier to deliver. In the overbooked case, overall utilization may be improved, although full quota utilization can no longer be guaranteed to all projects.

There are additional situations that may prevent projects from spending their allocations, such as adding projects to an already fully booked environment or resource down-time/outage. Different dynamic pricing models* could potentially be used to resolve such situations. For example, a dynamic pricing scheme could be employed where resources cooperate to balance the actual capacity with the circulating credits by adjusting the exchange rate. Although the challenge lies in designing such a credit-capacity balancing price scheme, it would be straight-forward to plug into JARM (which supports custom pricing schemes, see Section 4.2) once it is available.

*SweGrid uses a static pricing scheme with a fixed exchange rate of one credit per wallclock CPU second.

Competitive, market-based pricing schemes have been proposed in the literature [40, 55, 5, 7, 33, 34] as a means to balance load between resources (by attracting users to lightly loaded resources with low prices and vice versa) and achieve service differentiation (users pay more to receive better QoS). One such approach was demonstrated in [44], with minimal impact on the middleware but with a completely different resource provisioning model (based on virtualization). See Section 6 for a continued discussion on market-based resource allocation.

4.2. Job Account Reservation Manager (JARM)

The Job Account Reservation Manager (JARM) integrates local resources into the VO-wide accounting context of SGAS. The globally scoped project allocations are enforced collectively by the JARMs, which intercept job submissions on the Grid resources and decide which jobs to grant access. Although the JARMs are coordinated by the bank, all job admission decisions are subject to local resource owner policies, which allows bank decisions to be overruled by JARM in order to honor the site autonomy of Grid participants. The modular structure of JARM allows it to interface with different combinations of middleware and scheduling systems and also allows custom JARM behavior to be implemented.

An illustration of JARM job submission handling is shown in Figure 2. An incoming job request to a Grid resource is intercepted by JARM through a pre-execution call-out from the Grid job submission software, in this case the GRAM component of Globus Toolkit 4. The call-out is handled by the reservation manager component of JARM, which:

- Finds an account for the job submitter. The account may be explicitly specified in the job request. Otherwise, JARM must search for an account in the Bank.
- Estimates the job cost, typically based on the requested wall clock time, but it may use any function incorporating different resource types.
- Acquires a reservation on the project account, corresponding to the estimated job cost. A soft-state approach is used for the reservation, which is time-limited and will be released on expiration.
- Depending on reservation outcome and local site policy, decides if the job should be allowed to execute.

In essence, the pre-execution call-out to JARM is an authorization decision based on the job submitter's credit balance. Assuming that the job was granted access, the workload manager then executes the job, typically queuing the job in the local scheduling system. When the job finishes, the workload manager makes a post-execution call-out to JARM, which notifies the commit manager of job completion. The commit manager then:

- Collects usage data for the job in the environment-specific format.
- Transforms it into a standard usage record and reports it to the LUTS.
- Based on job usage, calculates the actual job cost (which may differ from the estimated cost) and charges the project account with the previously acquired reservation. Any residual reservation amount is released.

As mentioned previously, SGAS performs soft enforcement of allocations in the sense that allocations are not necessarily enforced in a strict manner. Rather, local site policies may allow quota-exceeding jobs to run since resources can overrule the bank (deny) decision. Permitting jobs to run in spite of

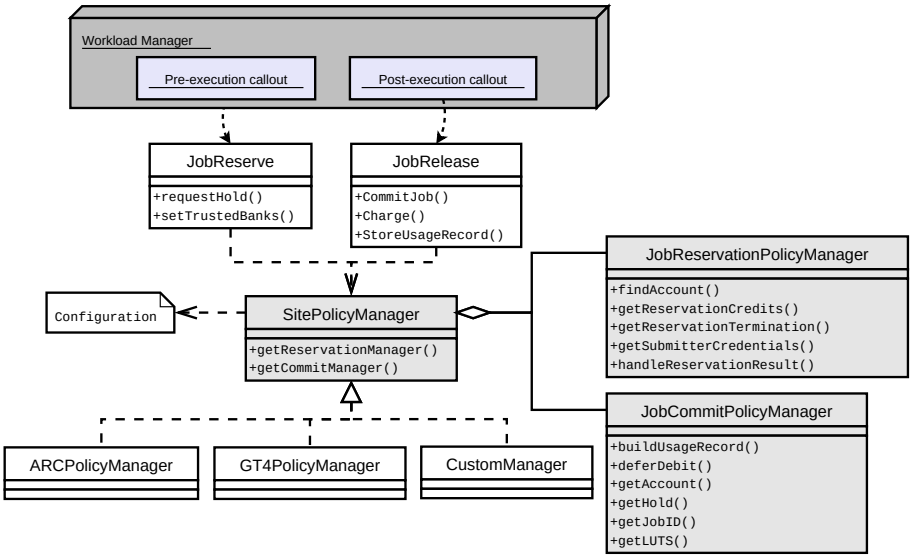


Figure 4. JARM components.

account overdraft represents a trade-off where utilization is increased at the expense of fairness. In case fairness is the sole objective, enforcement should strictly disallow all quota-exceeding jobs. Too strict enforcement of quotas may, however, lead to poor utilization where the unused capacity of inactive projects could have been used by other projects.

As a fault tolerance measure, to guarantee operation in the event of bank failures, JARM policies on the resource may allow jobs to run even though the bank is unreachable. In such cases, no reservation is acquired and the job is only logged in the LUTS on completion. An administrator can make corrective job debits when bank contact has been reestablished.

4.2.1. Integration Overview

Figure 4 illustrates the JARM components and the integration with the underlying workload manager, which has two distinct parts – a pre-execution and a post-execution call-out, corresponding to the reservation and charging activities respectively. The pre-execution call-out, which finds the project account, determines the job cost and requests an account reservation, may refuse a job, in order to prevent overdrafts and enforce fairness. The post-execution call-out collects usage data from the underlying scheduling system, charges the reservation and logs usage with the LUTS.

These call-outs use two pre-defined classes (JobReserve and JobRelease) to take care of all bank and logging service interactions. The behavior of JobReserve and JobRelease can be customized by a configurable SitePolicyManager (Java interface) implementation, which acts as a plug-in point for

site-specific behavior. The workload manager call-outs must provide sufficient runtime context (such as credentials, job description, local user id, etc) to enable the site policy manager decisions. Custom site policy managers typically need to be provided for each underlying middleware. The site policy manager has two components: a reservation manager which takes care of job cost calculation (that is, the reservation amount), can provide custom mechanisms for finding a chargeable account, and may implement specialized failure handling (for example, to log faults); and a commit manager which collects usage data, builds a usage record, and provides hold and logging service references to the JobRelease class, which is responsible for committing (charging and logging) a job.

4.2.2. Integration approaches

Different approaches are conceivable for the pre-/post-execution call-outs. The NorduGrid/ARC [47] and GT4 GRAM integration uses two different approaches. These can be categorized as plug-in script and interceptor approaches, respectively.

SGAS integration with ARC is performed via a mechanism that allows configuration of authorization scripts that are to be executed by the ARC job submission software at state transitions during the job life-cycle. ARC can thus be configured to execute a reservation script when the job is accepted and a charging script when the job reaches its finished state.

The GRAM integration is performed by means of SOAP message interception. As shown in Figure 5, JARM is integrated using one interceptor for request messages and one for response messages. The first handler, which is the pre-execution reservation call-out, transparently intercepts inbound job requests to the createManagedJob operation of the ManagedJobFactoryService (MJFS). This is done by inspecting all incoming messages. Depending on the bank response and local JARM policy configuration, the handler may choose to refuse the job request, by raising an authorization exception. If the reservation is successful the request is passed along to the MJFS which creates a WS-Resource representing the job.

A second message handler then intercepts the outbound response messages from MJFS. This message handler digs out the job WS-Resource EPR from the response SOAP body, and then establishes a subscription (using the WS-Notification [24] specification) with the resulting job, before the response message is transmitted. Later, when the job finishes, a notification is sent, and the post-execution call-out can be invoked to charge the job. Since GRAM does not provide a uniform usage format for the different scheduler adapters, usage data needs to be collected from scheduler logs. Therefore, the GRAM integration provides plug-in support for adding scheduler-specific usage data collectors. The implementation provides two built-in usage data collectors for the Fork and PBS scheduler adapters. The GRAM interception approach to integration is actually quite generic and could be used to charge for arbitrary service invocations (where MJFS is replaced by any other service).

Although quite different, both of these integration approaches share a common characteristic: they are non-intrusive in the sense that no workload manager code needs to be modified for the integration.

4.3. Logging and Usage Tracking Service (LUTS)

The Logging and Usage Tracking Service (LUTS) provides a Web service interface for publishing of usage data in the uniform format prescribed by GGF-UR, and for query-based retrieval of usage data using the XPath query language.

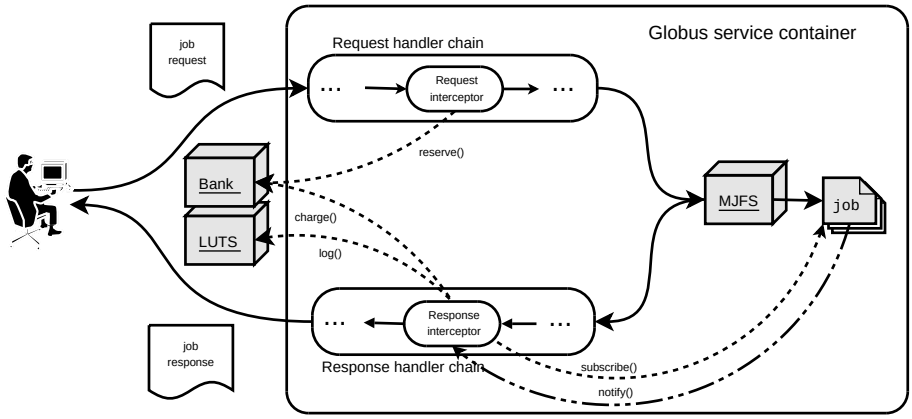


Figure 5. WS-GRAM integration.

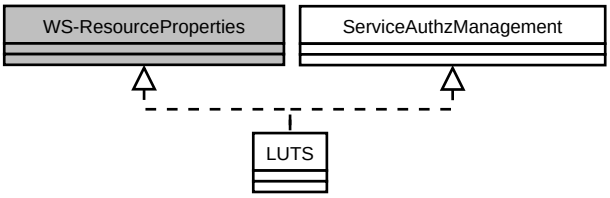


Figure 6. Logging and Usage Tracking Service (LUTS) interface.

The service interface, shown in Figure 6, is simple and defines no operations of its own. Instead it relies on the document-centric operations provided by the WS-ResourceProperties portTypes. LUTS employs the same security infrastructure as the rest of SGAS, including the ServiceAuthzManagement rights administration interface, allowing differentiated (more restricted) publish and (more generous) query access rights.

SGAS uses a native XML-database (eXist [16]) for persistent storage and recoverability of usage records. We provide a query dialect, implemented as a custom query expression evaluator [20], allowing XPath database queries to be executed through the Web service interface. The expression evaluator redirects the embedded XPath query to the back-end XML database, effectively exposing a database view through the service interface. The same mechanism is used to query account transaction logs.

The extension-points defined in the usage record specification in concert with the schema-agnostic LUTS storage of usage records (usage record documents are stored “as is” in the XML database)

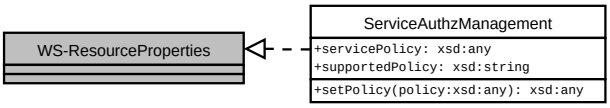


Figure 7. ServiceAuthzManagement (SAM) service interface.

facilitate extensibility, which allows sites to publish custom usage record elements, without modifying the LUTS.

4.4. Security

SGAS offers a flexible and interoperable end-to-end security solution through the use of standard security mechanisms and a fine-grained, highly customizable authorization framework.

All SGAS components share the same security infrastructure, based on standard privacy and integrity message protection supporting both message-level security (via WS-Security [36], WS-SecureConversation [28]) and transport-level security (via TLS [9]), as provided by GT4.

Another part of the SGAS security solution is a service-orthogonal authorization framework (described in more detail in [43]) that allows run-time administration of XML-based authorization policies through a Web service interface (ServiceAuthzManagement, shown in Figure 7), with plug-in support for different back-end authorization engines. The current back-end is based on the eXtensible Access Control Markup Language (XACML) [23] and allows fine-grained access policies to be set up for services and WS-Resources.

Finally, SGAS uses credential delegation and single sign-on to allow JARM to transparently reserve allocation and charge usage on behalf of the user. This leads to an “in-blanco” trust model where users must trust that the resource targeted for job submission is well-behaved (and does not abuse the account). Resource owners may configure a set of trusted banks with JARM in order to prevent users from circumventing allocation enforcement by supplying a bogus account in a fake bank (through the job description). The bank does not need to be configured to explicitly trust resources, since resources always act on behalf of users (using delegated credentials).

4.5. Service Distribution and The Virtual Bank

SGAS is intended to cope with Grid environments of differing sizes, ranging from departmental- to organizational-, national-, and international-size Grids. Hence, SGAS needs to scale with user populations and resource collections of increasing size, and handle the accompanying increase in data traffic and request load without becoming overloaded.

The solution is to virtualize the bank by partitioning it across several branch servers to allow for dynamic (and transparent) provisioning of additional load-balancing servers to adapt to VO growth, while still presenting the bank as a single logical service. The key enabler of the virtual bank is an abstract (location-independent) naming scheme, where each account is assigned a logical name that

is dynamically resolved by clients into the network endpoint address of the physical server where the account resides.

The virtualization-enabling naming scheme is supported through a service infrastructure for registration and resolution of name-to-address mappings. These services, which we collectively refer to as a name service, are not SGAS-specific, and could thus be useful outside the SGAS context as well.

Besides simplifying the lives of end-users, who can use abstract names like sgas://atlas-account instead of physical addresses, the location-independent naming scheme also produces scaling-, migration-, and location-transparency.

4.5.1. Name Registration and Resolution Service

The name service stores logical references, which map location-independent names (Uniform Resource Identifiers, URIs) to network endpoints, and translates the logical references into their physical endpoint addresses. The name service is a logical service that is composed of two separate sets of Web services: one for name registration, and one for name resolution.

Name registration. The registration part of the name service manages hierarchically arranged collections of transient, many-to-many name-to-address mappings. It is inspired by the Resource Namespace Service (RNS) specification [39], and exposes a WSRF-based registration interface. Note that we did not implement RNS as is, in part since the specification was in a state of flux at the time, and in part since it did not exactly match our needs.

For resolution, the Resolver portType of the OGSA-defined WS-Naming [26] specification has been implemented. Resolution produces WS-Names, an EPR that has been augmented with abstract name and resolver fields (exploiting the extensibility points in WS-Addressing [27]). The service interfaces that constitute the name service are shown in Figure 8.

For the registration part of the Name service, there are three separate Web service interfaces. Each logical reference is represented by a WS-Resource that can be accessed via the LogicalReference Web service. Each mapping, associating a logical reference with an endpoint, is modelled via a Mapping WS-Resource. The LogicalReferenceFactory implements the WS-Resource factory pattern, allowing new logical references to be created and existing logical references to be rebound to new endpoints, to allow for transparent server migrations.

Mappings are short-lived resources (based on WS-ResourceLifetime [49]), and their lifetime needs to be renewed to prevent removal. Besides facilitating “self-cleaning” registry behavior, this soft-state approach also simplifies client-side invalidation of resolution caches by providing an explicit validity time for each mapping. The notion of parent mapping introduces an additional mechanism for controlling mapping lifetimes. A mapping may live within the context of a parent mapping, meaning that the child mapping only exists as long as its parent mapping is “alive”. Thus, mappings can be organized into parent-child hierarchies. Together, lifetime and parent referencing allow joint lifetime handling for several mappings through a common parent mapping, which improves overall scalability by reducing the number of mapping renewal invocations. For example, this approach is used in the virtual bank to renew the lifetimes of all branch account mappings via a single renewal of the branch mapping, which is the parent of all account mappings.

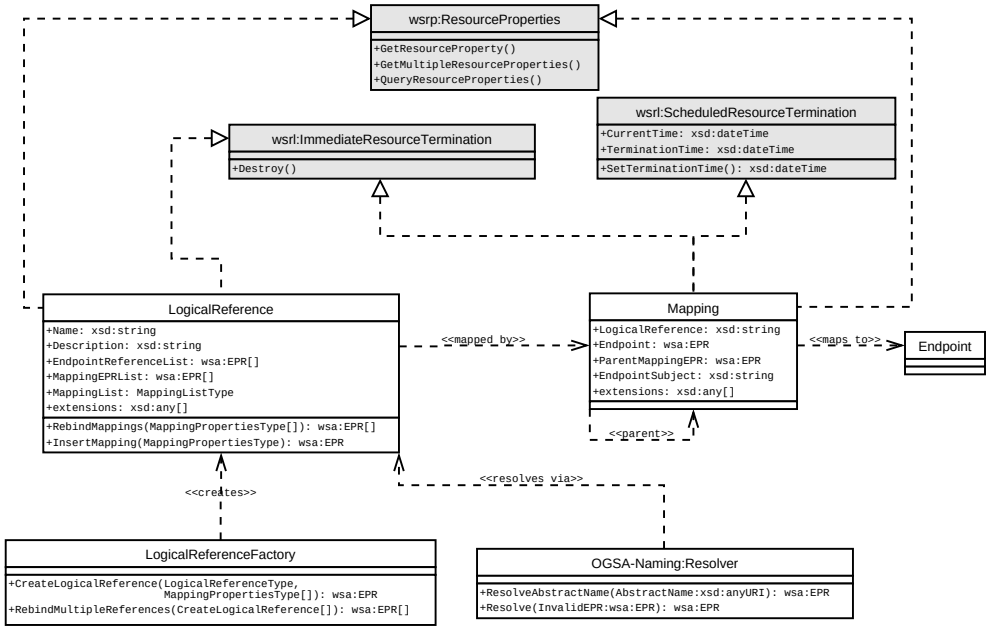


Figure 8. The Name Service service interfaces.

Mappings may be annotated with an extensible set of properties, allowing them to carry additional data besides pure addressing information. A mapping may, for example, contain the X.509 identity of the server hosting the mapped endpoint, which provides a mechanism for dynamic trust establishment. Given that a client trusts the resolver to supply valid mappings, trust can be dynamically established with the resolved endpoint prior to invocation (the client adds the identity of the resolved endpoint to its set of trusted subjects). This simplifies client security configuration which only needs to recognize the name service as the single point of trust in the system.

Name Resolution. The OGSA-Naming Resolver service provides two operations: one for translating abstract names (URI) into a WS-Name, and one for renewing an invalid WS-Name (supporting fail-over from obsolete mappings). Although the WS-Naming specification prescribes the use of universally unique identifiers (for example, UUIDs) for WS-Name abstract names, this requirement is relaxed in the SGAS environment, where abstract names only need to be unique within the context of a single virtual bank, hence facilitating user-friendly names.

SGAS provides a client-side abstraction layer that performs automatic name resolution (and EPR renewal if necessary), hiding all resolution interactions from the developer, allowing clients to use abstract names, such as sgas://account, as regular service addresses and transparently have endpoint resolution and trust establishment carried out by client-side SOAP message handlers.

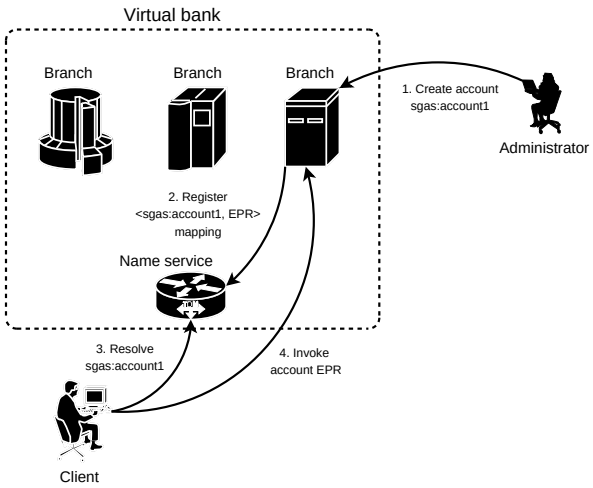


Figure 9. A virtual bank distributed across several hosts.

Scalability. Several measures have been taken to ensure scalability. First, client-side caching of resolution results may be used quite aggressively. Second, mapping registration renewals may be performed on a per-server basis (rather than on a per-mapping basis) through the combination of mapping lifetime and parent-child mapping relationships. Third, the solution lends itself to the creation of ("DNS-style") hierarchies of resolution services, that when combined with caching schemes off-load "top-level" name services. Fourth, the amount of update traffic is reduced by the use of batch operations (e.g. rebindMappings) that group together several actions in a single operation.

4.5.2. Virtualizing the Bank

In the distributed virtual bank configuration, each bank becomes a bank branch responsible for a separate subset of accounts. This is illustrated in Figure 9, which shows a virtual bank that is distributed over three distinct branch servers. The figure also shows how a bank administrator creates a new branch account (named sgas:account1) which results in a name to physical endpoint reference mapping being added to the virtual bank name service. The account can then be contacted through its abstract name, which is resolved to its physical address through the name service prior to invocation.

The name service, being the abstraction layer that hides the internal details of the virtual bank, manages name-to-address mappings for the branch servers. When an account is created, the branch will register a mapping between the logical account name and its physical address in the name service. The name service enforces a name uniqueness constraint across the branches to prevent duplicate account names.

All branches register their presence in the virtual bank by adding a branch mapping to the "root logical reference" of the virtual bank. This root reference is resolved into the set of branch servers by JARM to perform bank-wide (branch-crossing) searches for accounts when left unspecified by the job submitter.

Whenever a branch is (re)started, it will register a branch mapping with the root reference and rebind all account mappings (through a batch-mode registration operation). The accounts are rebound to overwrite obsolete bindings in case the branch has migrated. Each account mapping is created as a child of the branch mapping. This allows the branch to periodically renew all account mappings by extending the lifetime of the branch mapping only. Unavailable branches will eventually disappear from the virtual bank when their branch mapping times out.

Adding a new branch server to the bank places no additional burden on resource owners. The resource (JARM) administrators simply configure the virtual bank name service as a trusted target. Trust with new branches will then be automatically established at resolution time.

Although it has not yet been attempted, a service distribution approach, similar to that of the virtual bank, could also be applied to distribute the LUTS, thereby supporting heavier traffic and handling of larger data volumes. However, for a complete view on Grid usage, such a solution requires distributed query processing as well.

4.6. Scalability and Stability Measures

The progress from prototype to production code has prompted us to take measures to enhance overall system stability and scalability to cope with the increased load and data traffic volumes that come with large scale production Grid environments. The virtual bank, described separately in Section 4.5, represents one such scalability measure.

In an effort to reduce bank and LUTS load and to reduce the amount of network traffic, JARM can be configured to spool job usage records and defer logging and account charging. Instead, these activities are performed periodically in batches. For this purpose, the LUTS and bank provide batch operations that allow a resource to register several usage records or debit multiple jobs in a single invocation.

Since the usage record repository of LUTS may contain large volumes of data and given that users may pose arbitrary XPath queries against the LUTS, result sets may contain more documents than can be transmitted in a single response message without consuming all client or server memory during message (de)serialization. To this end, a cursor-like mechanism has been implemented to allow incremental fetching of large result sets in several smaller messages. In this implementation, a query evaluator responds to a query by returning a reference to an Iterator WS-Resource on the server that represents the cursor position in the result set. The client can traverse the result set segment by segment by making repeated invocations to the Iterator. The Iterator WS-Resource has bounded lifetime (via WS-ResourceLifetime [49]), which allows result set data and iterator state to be discarded automatically. Our Iterator construct can roughly be seen as a WSRF-based counterpart of WS-Enumeration [1]. In a similar manner, SGAS supports partitioned uploading of usage record batches to the LUTS to allow a large set of (spooled) usage records to be published piece by piece.

Since both the bank and the LUTS are supposed to run without interruption over long periods of time, stability is a key requirement. A lot of effort has gone into stabilizing the server memory consumption. To reduce the memory footprint, resources are checkpointed to persistent storage and held in memory-sensitive caches (based on Java soft references). We have also improved memory

management via periodical sweeper tasks that are triggered when configurable memory watermarks are reached to reclaim memory, e.g. by pruning database caches and secure contexts. For purposes of recoverability, backups of the XML database used by SGAS services are periodically written out to disk. This is done at runtime to prevent service interruption to clients. To cope with the ever-growing nature of logs and prevent them from growing too large, periodical log rotations of the LUTS repository and account transaction history are performed that write out old entries from the database to the file system.

During this work, we noticed the recurring need to schedule server tasks for periodical execution (database backups, log rotations, sweeper tasks, etc). Therefore, we developed a general framework for scheduling the execution of service tasks, which can be seen as a server-side cron mechanism. The container administrator sets up individual time schedules for execution of tasks through a configuration file. The set of tasks and their schedules may be reconfigured at runtime. Besides the pre-defined schedules (monthly, weekly, daily, hourly, minutely, one-shot) and tasks, custom Java schedule and task classes can be added dynamically.

5. PERFORMANCE EVALUATION

This section presents a performance evaluation of the SGAS software, based on a number of real-world tests performed against different configurations of a local Grid testbed. The tests measure the overall system performance of SGAS, reveal scalability limits, and assess the performance impact of SGAS on the underlying job submission software. The tests have been performed on a Grid with GT4 as the underlying middleware (with the Web services-based GRAM workload manager).

5.1. Testbed

During the tests two separate sets of computers were used, both connected through a 100 Mbit/s campus network. The first set of computers were used exclusively for server-side components (to host SGAS services and the GRAM service container) and consisted of four computers, each equipped with a 2.0 GHz AMD Opteron processor, 1 MB cache memory, 2 GB internal memory and running the Ubuntu 5.10 Linux distribution. The second set of computers were used to launch test clients that issued bank requests or submitted jobs to the GT4 server. These computers all ran under the Debian Linux operating system and sixteen of them were equipped with Intel Pentium 2.8 GHz processors and 1 GB memory, while the other sixteen had AMD Athlon 64 2.0 GHz dual core processors with 2 GB memory.

The DiPerf performance testing framework [12] was used to drive and coordinate the test clients, collect time measurements, and compile the results into a single (global) timeline. Since all system clocks were synchronized via NTP and all hosts connected to the same network, clock synchronization was kept tight (roughly within 1 ms) between the computers, and the impact on time measurements is therefore negligible.

Two slightly different versions of SGAS were used in order to investigate what performance gains could be achieved by reusing secure network connections. The “regular” SGAS-version is based on Java WS Core of GT4.0.2, and the other, which we refer to as the “connection reuse” version, is built

with a CVS snapshot of Java WS Core dated 9/15/2005 that supports persistent HTTPS connections[†]. The connection reuse SGAS version features an improved JARM that caches recently used account stubs and reuses previously established HTTPS-connections, thereby avoiding the multi-roundtrip performance penalty involved in the initial TLS connection establishment handshake. Unless stated otherwise, the tests use the regular SGAS version.

Since GT contained some code that seriously hampered throughput in our multi-client tests, we rewrote GT4.0.2 by introducing a small piece of GRAM code that caches user home directories, rather than having a perl script re-evaluate it on every use (which is expensive in Java). This change significantly cut the time spent by threads in a critical section of the GRAM code, resulting in almost a factor six throughput increase (in "streamlined" submission mode). The patch has also been applied to the GRAM code in the Globus CVS.

5.2. Experiments and Performance Results

There are several aspects of performance that deserve an in-depth study, however, we have narrowed it down to three test cases. First, we have tested account reservations against the bank. This is the single most important bank interaction, as it lies on the job submission critical path. Second, we illustrate the scalability improvements achievable with the virtual bank solution. Third, we have investigated the SGAS performance impact on the underlying workload manager (GRAM). In these experiments, we focus on three metrics:

- Response time: the client perceived (end-to-end) request time. That is, the time (in ms) from sending a request until receiving the result.
- Throughput: the rate of handled requests, measured in the amount of completed requests per minute.
- Load: the number of active (request-issuing) test clients at any instant, measured in number of clients.

5.2.1. Account Reservation Test

The account reservation test shows the delivered bank performance under heavy load. It simulates the scenario of an escalating "job storm", where the job submission rate (and hence, reservation request rate) steadily rises, reaches a peak and finally starts to drop. In this test, all client machines are launched against the bank. A new test client is started every 60 seconds, and each client issues requests for 40 minutes. The target account is chosen at random from the accounts hosted by the bank.

The results are shown in Figure 10, where response times, throughput and load curves are superimposed in the same diagram, to allow for easy correlation. The (stair-shaped) load curve values are read from the right axis, whereas response times and throughput values are read from the left axis.

As the figure shows, new clients are continuously started for 30 minutes. By that time all 30 clients are active and the job storm reaches its climax. As concurrency increases, throughput grows as long as there is spare server capacity to handle the additional load. However, after roughly 16 minutes a

[†]This functionality is provided as part of the GT4.1.0 development release.

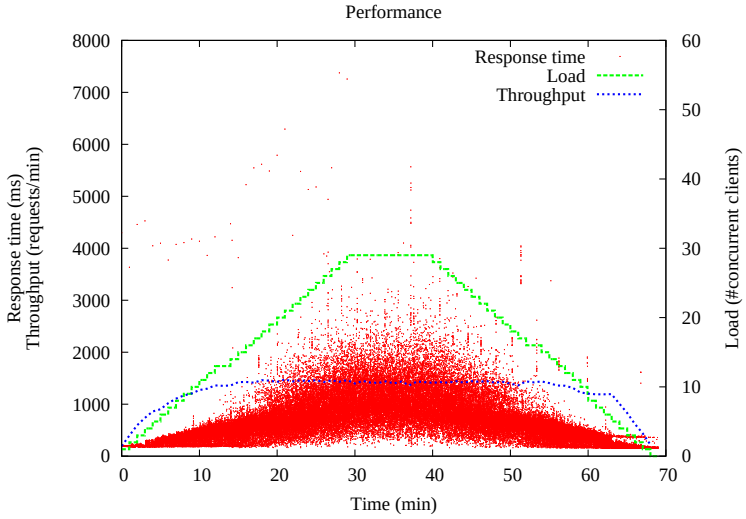


Figure 10. Reservation requests against the bank.

throughput limit is reached (about 1400 requests/min). After that limit has been reached the throughput remains unchanged as further clients are added, while individual clients start experiencing a significant increase and variation in response times. The opposite effect can be observed after peak load has been reached. As load decreases, response time fluctuation and average response time are reduced, and with approximately 15 minutes left of the test, load has been sufficiently reduced to bring throughput down from the upper limit and start falling towards zero as more and more clients complete.

Under light load, response times are roughly 200 ms whereas extreme response times of several seconds can be observed during peak load. From this test, we can conclude that in the given environment, the bank is able to handle a peak load of approximately 1400 reservation requests per minute.

During the test a total of 86584 reservation requests were issued. Given the test duration of 70 minutes this yields an average request handling time of about 49 ms, which is a factor four improvement over the 200 ms taken to execute a single serial request. The reason is that much of the response time can be attributed to message transmission (client-side message processing and secure connection establishment) allowing for a rather high degree of request concurrency. To sum up, the bank reaches its throughput limit at 1400 requests per minute, or about 23 requests per second. Such load would correspond to a Grid environment where 23 new jobs are submitted every second. A Grid with that kind of job turnover must either be large, or serve users with extraordinarily short jobs. To scale with even larger environments a virtual bank can be deployed, as illustrated in the next test.

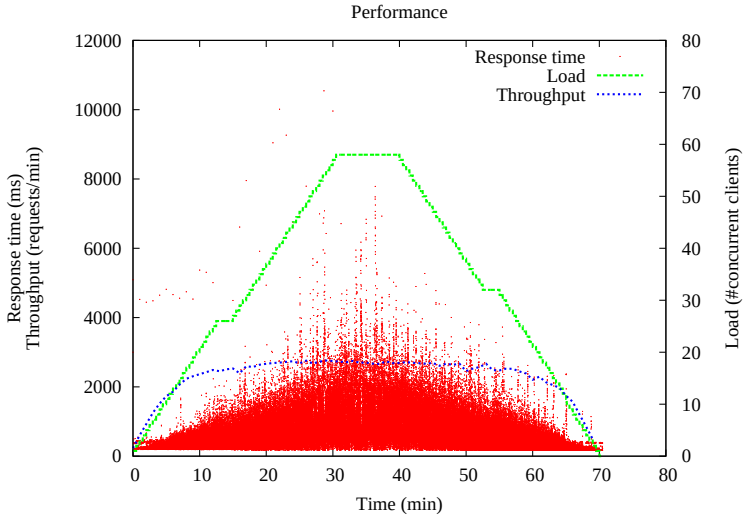


Figure 11. Reservation requests against a virtual bank with two branches.

5.2.2. Virtual Bank Test

The test setup for the virtual bank is similar to that of the account reservation test, apart from that twice as many clients are launched (two from each client machine) with twice as high rate (a new tester is introduced every 30 seconds). The bank is configured as a virtual bank with a name server and two branch servers. The accounts are evenly divided between the two branches.

During the test, each tester makes a reservation against a randomly chosen account. This includes resolving the account name with the name service to translate the abstract account name into its physical address. The collected results are shown in Figure 11, with axes as described in Section 5.2.1.

We can see that the maximum throughput in the virtual bank case is reached after about 20 minutes (40 clients). The throughput limit in this case is roughly 2700 requests/min, which is close to a doubling of throughput from the single branch case (1400 requests/min). The fact that we do not achieve a throughput doubling, is most likely an effect of imperfect load balance between the two branch servers. Clients pick a target account at random, which makes a certain degree of load imbalance at any instant quite likely. A closer inspection of the figure shows that some of the initial requests take about twice the time of others. This is caused by the name lookup, which requires an extra roundtrip in order to contact the resolution service, and this invocation roughly doubles the response time. Client-side caching of name resolution results allows subsequent resolutions to be made against the local cache. Since branches operate completely independently from each other, the two branches are capable of handling twice the load of a single bank, with twice as high throughput. Similar improvements should be observed for each added branch, given that load is sufficiently well balanced between the branches.

Table I. The submission options.

Submission Option	Description
Batch mode (**B)**	Waits for job to be accepted.
Interactive mode (**I)**	Waits for job to complete.
Per-job delegation (**P)**	Delegates a separate proxy for each job.
Shared delegation (**S)**	Delegates a single proxy shared by all jobs.

5.2.3. SGAS Performance Impact Tests

The following tests measure the performance impact of SGAS (or rather the account reservations) on job submission throughput in the GRAM workload manager. The tests all follow the same general pattern: eight simultaneously started tester clients submit jobs to a GRAM container for a 20 minute period. Hence, the load is kept constant throughout the test. Each tester submits /bin/true Fork job requests to the GRAM container, one at a time, always waiting for an operation to complete before starting the next. We have deliberately chosen a minimalistic job (quick turnaround, no file staging, no batch system access, etc.) in an attempt to isolate those parts of GRAM that are affected by JARM. For more advanced job submissions, the relative performance impact of JARM is greatly reduced.

Three different GRAM configurations and four different submission modes were used to compare different setups and cover the different job submission styles that end-users may prefer. For the GRAM configurations an unaltered "accounting-disabled" container was tested for reference, while two "accounting-enabled" containers (with JARM intercepting jobs) were tested: one with the regular SGAS version and one with the connection reuse version. The four tested submission modes result from combining either interactive or batch mode submissions with either per-job or shared credential delegation. These job submission options are explained in Table I, which also shows the abbreviation used for each option (B,I,P,S). For the accounting-enabled tests, a bank was located on a separate machine, hosting accounts that were chosen at random by clients to charge jobs.

A summary of the protocol steps involved in each of the job submission modes is shown in Table II. The *delegate proxy* step involves delegating a user proxy to the Delegation Service of the targeted GRAM server. The *submit job* step only requests the execution of a job, it does not wait for execution to finish. In interactive mode, clients wait for jobs to complete by subscribing to *state notifications* which are sent to the client on job state transitions, typically when the job reaches the active, cleanup and done state. *Job cleanup* and *proxy cleanup* refers to a destroy invocation against the WS-Resource representing the job or proxy in question. Note that in batch mode tests, a completed request only means that GRAM has accepted the job. Hence, at the end of the test there may be several jobs that are still awaiting execution. Also note that the shared delegation submission mode delegates a proxy prior to submitting the first job.

Figure 12 shows the "streamlined" job submission mode test case for our three GRAM configurations. Streamlined in this case refers to the minimal sequence of GRAM protocol steps that this submission mode uses. Each client delegates a single proxy credential that is shared by all jobs

Table II. GRAM protocol steps.

Protocol step	B/S	B/P	I/S	I/P
Delegate proxy		√		√
Submit job	√	√	√	√
Execute job			√	√
State notifications			√	√
Job cleanup			√	√
Proxy cleanup				√

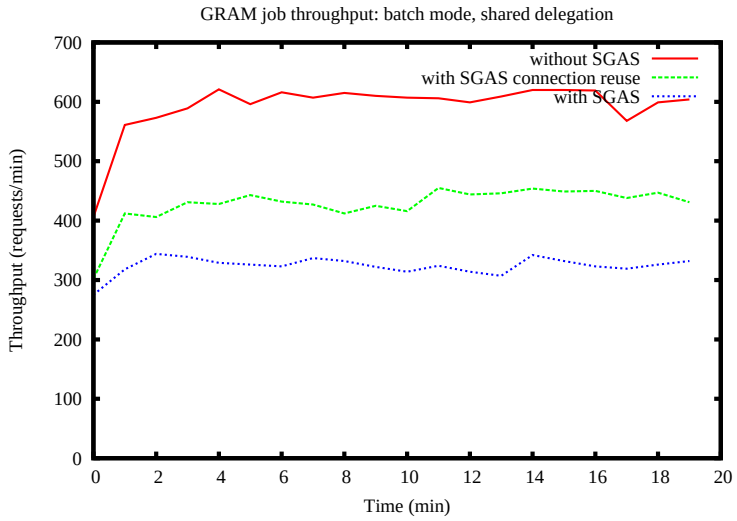


Figure 12. Streamlined (batch mode, shared delegation) job submissions.

(shared mode) and no job state change subscriptions are established (batch mode). As can be seen from the other GRAM figures (13, 14, 15), this test case clearly shows the highest throughput numbers (it outperforms the other tests by a factor six). As a consequence, this is the test case where SGAS incurs the largest overhead on GRAM submissions. The unaltered GRAM amounts to 590 requests per minute on average, whereas the regular SGAS integration reduces the throughput to half (320 requests per minute). The connection reuse SGAS version offers a significant improvement in this case, handling on average 430 requests per minute. The main source of SGAS overhead is the account reservation invocation performed prior to accepting each job submission.

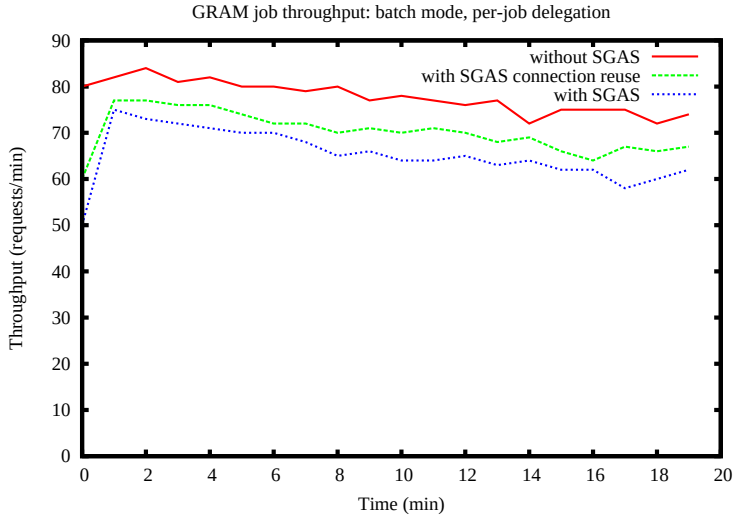


Figure 13. Batch mode, per-job delegation job submissions.

Note that, although the throughput drop with SGAS is quite significant, this test case puts extreme pressure on the GRAM server. A job submission rate of ten arriving jobs per second is unlikely to be observed on a real production environment Grid resource, and hence the main merit of these throughput numbers lies in establishing an upper throughput limit for GRAM (with and without SGAS). We conclude that neither the unaltered GRAM nor the "accounting-enabled" GRAM container is likely to become a bottleneck in realistic job submission scenarios. Note also that the bank only is lightly loaded in this test with a few hundred reservations per minute, which is far from overloading the bank, which does not reach its peak throughput until 1400 requests per minute.

Figure 13 illustrates the throughput numbers when per-job delegation has been added to the streamlined job submissions of Figure 12. As can be seen by comparing the figures, the additional protocol step introduced by delegating a proxy credential with each job seriously hampers throughput which is down to 78 jobs/min without SGAS, 65 jobs/min with SGAS and 70 jobs/min with the connection reuse SGAS version. As expected, the relative throughput difference between the accounting-enabled and accounting-disabled GRAM is reduced as a result of the SGAS performance impact becoming a smaller relative factor as additional protocol steps are added.

Results for the "all-inclusive" submission mode are shown in Figure 14. Here, clients delegate separate proxy credentials for each job (per-job delegation) and also awaits completion (interactive mode) for each job. It is the test case that includes most protocol steps and, consequently, shows the lowest throughput numbers with 56 jobs/min in the pure GRAM case, 50 jobs/min for SGAS-enabled GRAM and 53 jobs/min for SGAS with connection reuse.

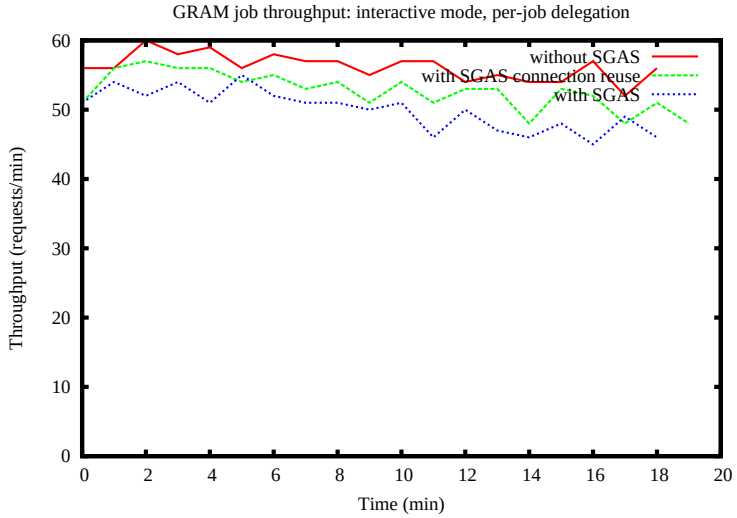


Figure 14. All-inclusive (interactive mode, per-job delegation) job submissions.

The steadily decreasing throughput curves that can be observed in the per-job delegation test cases (Figure 13 and 14), may indicate scalability issues in handling of large amounts of job submissions with per-job credential delegation. Furthermore, the similarity of the curves in all test configurations indicates that SGAS is not the cause of the problem. Rather, the problem is likely to be found within the GRAM or the Delegation Service code.

The throughput numbers when interactive submission mode is added to the streamlined case are shown in Figure 15. In these tests, where clients wait for each job to complete and cleans up the job, the SGAS-disabled GRAM manages to handle 95 jobs/min, whereas the SGAS-enabled GRAM manages 84 jobs/min. In this case, the connection reuse SGAS version case is very close to the unaltered GRAM with 93 jobs/min. Judging by the sheer number of message exchanges involved in the different submission modes, one would assume that the B/P mode (Figure 13) would allow higher throughput than I/S mode (Figure 15). As our results show this is not the case. From this observation, we conclude that proxy delegation is a heavyweight operation, which should be avoided whenever possible (for instance, by using the shared delegation approach).

From the figures we can see that as more GRAM protocol steps are added to the job submissions (interactive and/or per-job delegation) the relative impact of SGAS on overall throughput becomes smaller (the SGAS throughput is closer to that of the unaltered GRAM), as a result of the SGAS overhead being shadowed by additional sources of overhead. For the same reason, the relative effect of connection reuse becomes less dramatic. Table III illustrates the overhead incurred by SGAS on GRAM, by showing the delivered throughput in percent of the unaltered GRAM throughput, for the

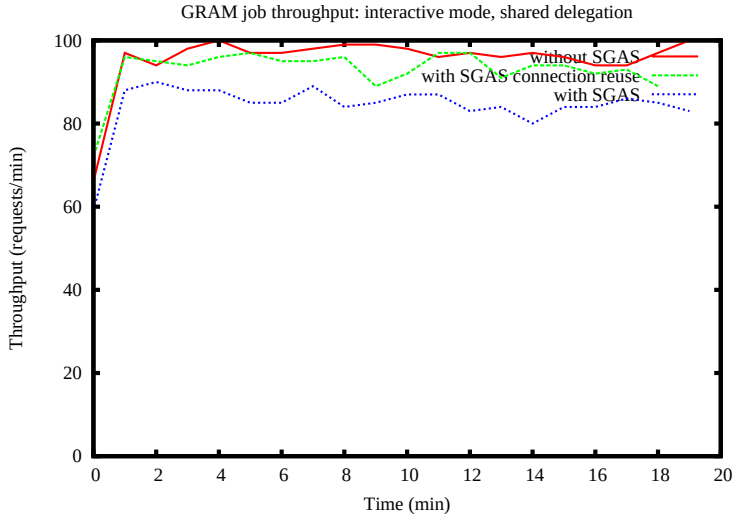


Figure 15. Interactive mode, shared delegation job submissions.

Table III. Delivered throughput in percent of normal GRAM throughput.

SGAS version	B/S	B/P	I/S	I/P
Regular	55 %	84 %	88 %	89 %
Connection reuse	72 %	90 %	97 %	94 %

different SGAS versions and job submission modes. For these numbers, aggregate throughput values were calculated over the entire duration of the test.

In summary, SGAS only incurs marginal overhead on GRAM job submissions (roughly around 10–15%). The exception is the batch-mode, shared delegation test case where the throughput reduction is quite significant (45% with regular SGAS and 28% with the connection reuse version). However, the batch/shared test case represents an extreme scenario with a job arrival rate of ten jobs per second, which is highly unlikely to be observed on a real-world Grid resource. Hence, we conclude that for all practical purposes neither GRAM nor SGAS threatens to become a limiting factor on the job submission handling capability of a Grid resource.

6. RELATED WORK

Compared to the widely distributed and heterogeneous nature of Grid environments, resource allocation and usage tracking in HPC/cluster computing environments is typically simplified by only accounting for local site usage on a set of homogeneous resources that run on a single platform, with a common security solution and that share a common data format for usage logging.

There are several resource allocation systems that target Grid environments. Some of these divide Grid capacity between users via a priori allocations in a manner similar to the way project allowances are awarded in SweGrid, while other systems let market forces allocate capacity via supply and demand driven interactions.

Gold [32] is a feature-rich, open-source accounting and allocation manager system that is similar to SGAS in operation and enforcement mechanism. It has a bank that manages project accounts with time-stamped allocations (implementing a "use-it-or-lose-it" policy), and also supports additional features such as nested accounts and user/machine-specific allocations. Gold is tightly integrated with the Maui scheduler and, in contrast with SGAS, not targeted towards simple middleware integration, which could be a barrier for adoption. Gold is the successor of QBank [31], which was aimed at single institution accounting.

The Distributed Grid Accounting System (DGAS) [10] is an accounting system developed within the context of the EGEE project [15], which besides pure usage tracking (metering and logging) functionality provides an optional layer of functionality for implementing economic models, which can be used to establish a resource market, driven by supply and demand, where provider organizations earn credits from users by executing user jobs. Earned credits may be redistributed over the organization users, leading to a zero-sum exchange of resources where total contribution is balanced between sites. For "economic mode" operation, DGAS is integrated with the Workload Manager resource broker of the LCG middleware [35], thus limiting its scope of applicability.

GridBank [4] is a bank service, developed within the GRid Architecture for Computational Economy (GRACE) architecture, to support economy-driven Grid interactions between resource consumers and providers. The GridBank service manages consumer and provider accounts, stores usage records, and handles payments between accounts. GridBank is targeted towards GT2 and provider-side deployment is quite intrusive, requiring modifications to GT2 job manager code. On the consumer-side, a GridBank payment module needs to be integrated with the resource broker (Nimrod/G) to forward a payment cheque to the resource that covers the job cost.

We note that SGAS distinguishes itself from the other accounting-related efforts, with its emphasis on middleware and scheduler independence and strong focus on interoperability, Web services and Grid standardization work.

APEL [6] is a usage tracking system that collects usage from Grid resources, much like the JARM and LUTS components do for SGAS. APEL parses batch logs to gather usage data and publishes it in the (non-standard) relational data format as prescribed by the R-GMA information system [41].

A share-based approach to allocate Grid capacity has been proposed by [11] and [13]. These solutions allocate resources by means of share policies that divide aggregate VO capacity between user groups, which are granted target shares of the total Grid capacity. These target shares are delivered using scheduling-based mechanisms. In [13] enforcement is carried out in a collective manner by the Grid resources via local (batch system) job scheduling with a global view on usage. A combination of global (resource broker) and local job scheduling is used in [11].

Economic approaches to resource allocation have received a lot of attention in Grid research. A common point in these approaches is the establishment of a Grid resource market, often referred to as a computational economy. When it comes to computational economies, we can distinguish utility and pay-per-use computing (which is part of the "real" economy) from market-based resource allocation within a Grid (which establishes an "artificial" market). We focus on the latter category, where market economic principles have been applied in a number of projects [40, 55, 5, 7, 33, 34, 3].

Some of the most common arguments for market-based resource allocation are that (1) dynamic pricing schemes can balance load across both resources (by attracting users to lightly loaded resources with low prices and vice versa) and time (encouraging users to use more resources during off-peak hours) to improve overall utilization [5, 40, 33, 7], (2) it promotes user-centric scheduling with per-task QoS differentiation [33, 5, 7], (3) market prices regulate resource supply and demand towards a state of market equilibrium where supply and demand is at balance [40, 55, 5, 3], and (4) markets operate in a decentralized[‡] and efficient manner, without need for centralized control [5, 3].

To date, most work concerning Grid economy has either focused on simulating computational markets or developing architectural frameworks that support the establishment of an economy [54]. However, research has mostly been silent or provided little guidance on automated pricing mechanisms to create stable and self-sustainable markets (where supply and demand are in equilibrium). In fact, Nakai and Van Der Wijngaart [37] conducted a thorough analysis of General Equilibrium (GE) theory, a theoretical foundation for claims (3) and (4) and they argue that such claims are not supported by the theory. They reach the conclusion that the GE theory fails to explain actual economies, let alone a compute resource economy. Market economy is not dismissed as a global scheduling solution but they remark that widely held beliefs, such as (3) and (4), of market efficiency are not warranted. Such beliefs are merely a product of everyday life observations of (real) markets and the perceived ease and efficiency by which they allocate resources.

We believe that the main appeal of market-based solutions is not the promise of reaching theoretical equilibrium states of high efficiency, but the focus on designing incentives into the software to avoid misuse and overuse by strategic users affecting the stability and health of the overall system. In short they provide an answer to the "tragedy of the commons" effect apparent in many of today's Grids. Such incentives are not at odds with the SGAS model, on the contrary the static pricing model currently employed in SGAS could easily be replaced by an incentive compatible pricing scheme according to market principles.

Still, many open questions remain within the area of market-based resource allocation. Some of these challenges are outlined in [46]. Although we believe that computational markets hold a lot of promise, we call for further investigation and comparison between market-based approaches and other Grid resource allocation mechanisms, both in terms of computational efficiency and allocation efficiency.

[‡]Agents acting in self-interest achieve global "welfare".

7. CONCLUDING REMARKS

Without usage regulation a Grid threatens to fall victim to the "tragedy of the commons". We address this problem with a Grid accounting system that offers overuse protection and differentiated usage guarantees in collaborative Grid environments by coordinating enforcement of Grid-wide usage limits. We have presented the operation context and role of the SweGrid Accounting System (SGAS) as a capacity allocation mechanism that mediates the conflicting needs of the system stakeholders. SGAS allows allocation authorities to divide the aggregate VO capacity between users in a fair manner and coordinate allocation enforcement across the Grid without sacrificing resource owner autonomy.

SGAS employs a credit-based model where Grid capacity is granted to projects via Grid-wide quota allowances that can be spent across the Grid resources, which collectively enforce these allocations in a soft, real-time manner. The use of time-stamped credit allowances, with a limited validity period, reduces the risk of imbalance between modelled capacity (credits) and actual capacity by continuously revoking surplus credits. At the same time, it constitutes a flexible tool that, e.g., allows allocation authorities to distribute quota over time to prevent contention on allocation period borders, and supports various allocation strategies that can trade off flexibility in utilization for fairness or closer credits-capacity correlation.

The SGAS design addresses key challenges of Grid environments (heterogeneity, scale, decentralized management, security) and is flexible with respect to the types of usage that is accounted for. To cope with the inherent heterogeneity of Grid environments, SGAS is agnostic to the underlying middleware and scheduling systems. To date, it has been integrated with GT4 and ARC, but we hope that the JARM description and our integration experiences may serve as a reference for future integration into other middlewares.

Measures to achieve system scalability include incremental handling of large data sets and virtualizing the bank service across several servers to balance load. The key enabler of the virtual bank is an abstract naming scheme, which adds an extra level of name indirection by introducing a generic name service that manages name-to-address mappings. Branch servers register an abstract, location-independent name for each account, which is resolved by clients into the physical network address of the account prior to invocation. Besides facilitating the virtual bank, the abstract naming scheme also produces scaling-, migration- and location-transparency.

We have evaluated the performance and scalability of SGAS by conducting an extensive set of experiments on a Grid testbed. These experiments reveal that the bank is able to handle a peak load of 1400 reservation requests per minute, which would correspond to a Grid scenario where 23 new jobs are submitted every second. Furthermore, the bank capacity can be scaled up even further by adding bank branches to the virtual bank which can offer a linear improvement in throughput and load handling capacity. Finally, we conclude that the overhead incurred by account reservations on job submissions is marginal and, in any realistic scenarios, not a limiting factor to the job throughput capacity of the job submission software.

ACKNOWLEDGEMENTS

We thank Catalin Dumitrescu, Lars Malinowsky, Ioan Raicu, Åke Sandgren, Johan Tordsson, and Björn Torkelsson for technical support and helpful comments.

REFERENCES

1. J. Alexander, D. Box, L. F. Cabrera, D. Chappell, G. Daniels, C. Kaler, D. Orchard, I. Sedukhin, M. Simek, and M. Theimer. Web Services Enumeration (WS-Enumeration), March 2006. http://www.w3.org/Submission/WS-Enumeration.
2. Apache Web Services Project - Axis, August 2006. http://ws.apache.org/axis.
3. O. Ardaiz, P. Artigas, T. Eymann, F. Freitag, L. Navarro, and M. Reinicke. The catallaxy approach for decentralized economic-based allocation in grid resource and service markets. *Applied Intelligence*, 25(2):131–145, 2006.
4. A. Barmouta and R. Buyya. A Grid Accounting Services Architecture (GASA) for distributed systems sharing and integration. In *IPDPS'03*, France, 2003.
5. R. Buyya, D. Abramson, and J. Giddy. An Economy Driven Resource Management Architecture for Global Computational Power Grids. In *The International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, June 2000.
6. R. Byrom, R. Cordenonsi, L. Cornwall, M. Craig, A. Djaoui, A. Duncan, S. Fisher, J. Gordon, S. Hicks, D. Kant, J. Leakec, R. Middleton, M. Thorpe, J. Walk, and A. Wilson. APEL: An implementation of Grid accounting using R-GMA. In *UK e-Science All Hands Conference*, September 2005.
7. B. N. Chun and D. E. Culler. Market-based Proportional Resource Sharing for Clusters. Technical Report Technical Report CSD-1092, Computer Science Division, University of California at Berkeley, January 2000.
8. J. Clark and S. DeRose. XML Path Langugage (XPath) Version 1.0, November 1999. http://www.w3.org/TR/xpath.
9. T. Dierks and C. Allen. The TLS Protocol Version 1.0, January 1999. http://tools.ietf.org/html/rfc2246.
10. Distributed Grid Accounting System (DGAS), August 2006. http://www.to.infn.it/grid/accounting/.
11. C. Dumitrescu and I. Foster. Usage Policy-Based CPU Sharing in Virtual Organizations. In *GRID '04*, pages 53–60, Washington, DC, USA, 2004. IEEE Computer Society.
12. C. Dumitrescu, I. Raicu, M. Ripeanu, and I. Foster. DiPerF: An Automated DIstributed PERformance Testing Framework. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 289–296, Washington, DC, USA, 2004. IEEE Computer Society.
13. E. Elmroth and P. Gardfjäll. Design and Evaluation of a Decentralized System for Grid-wide Fairshare Scheduling. In *e-Science 2005: First International Conference on e-Science and Grid Computing*, pages 221–229, Washington, DC, USA, 2005. IEEE Computer Society.
14. E. Elmroth, P. Gardfjäll, O. Mulmo, and T. Sandholm. An OGSA-Based Bank Service for Grid Accounting Systems. In *Applied Parallel Computing*, Lecture Notes in Computer Science, pages 1051–1060. Springer-Verlag, 2006.
15. Enabling Grids for E-sciencE, August 2006. http://www.eu-egee.org/.
16. eXist – Open Source Native XML Database, August 2006. http://exist.sourceforge.net/.
17. I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing*, pages 2–13. Springer-Verlag LNCS 3779, 2005.
18. I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, W. Vambenepe, and S. Weerawarana. Modeling Stateful Resources with Web Services, 2004. http://www-128.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf.
19. I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200 – 222, 2001.
20. J. Gawor and S. Meder. GT4 WS Java Core Design, November 2004. http://www.globus.org/toolkit/docs/4.0/common/javawscore/developer/JavaWSCoreDesign.pdf.
21. Global Grid Forum, August 2006. http://www.ggf.org.
22. Globus: WSRF - The WS-Resource Framework, August 2006. http://www.globus.org/wsrf/.
23. S. Godik and T. Moses. eXtensible Access Control Markup Language (XACML) Version 1.0, February 2003. http://www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf.
24. S. Graham and B. Murray. Web Services Base Notification 1.2 (WS-BaseNotification), June 2004. http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-03.pdf.
25. S. Graham and J. Treadwell. Web Services Resource Properties 1.2 (WS-ResourceProperties), June 2004. http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-04.pdf.
26. A. Grimshaw and D. Snelling. WS-Naming Specification, September 2006. https://forge.gridforum.org/sf/projects/ogsa-naming-wg.
27. M. Gudgin and M. Hadley. Web Services Addressing - Core, December 2004. http://www.w3.org/TR/2004/WD-ws-addr-core-20041208/.
28. M. Gudgin and A. Nadalin. Web Services Secure Conversation Language (WS-SecureConversation), February 2005. http://specs.xmlsoap.org/ws/2005/02/sc/WS-SecureConversation.pdf.
29. G. Hardin. The tragedy of the commons. *Science*, 162(3859):1243–1248, December 1968.
30. D. Jackson, Q. Snell, and M. Clement. Core Algorithms of the Maui Scheduler. *Lecture Notes in Computer Science*, 2221:87–102, 2001.

31. S. Jackson. QBank – Allocation Manager, 2006. http://sss.scl.ameslab.gov/qbank.shtml.
32. S. Jackson. The Gold Accounting and Allocation Manager, 2006. http://sss.scl.ameslab.gov/gold.shtml.
33. K. Lai. Markets are Dead, Long Live Markets. Technical report, HP Labs, Palo Alto, CA, USA, February 2005.
34. K. Lai, L. Rasmusson, E. Adar, S. Sorkin, L. Zhang, and B. A. Huberman. Tycoon: an Implemention of a Distributed Market-Based Resource Allocation System. Technical Report arXiv:cs.DC/0412038, HP Labs, Palo Alto, CA, USA, December 2004.
35. LCG Middleware, August 2006. http://lcg.web.cern.ch/LCG/activities/middleware.html.
36. A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. Web Services Security: SOAP Message Security 1.0, March 2004. http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.
37. J. Nakai and R. F. Van Der Wijngaart. Applicability of Markets to Global Scheduling in Grids. Technical Report NAS Technical Report NAS-03-004, NASA Advanced Supercomputing (NAS) Division, February 2003.
38. Open Grid Services Architecture WG (OGSA-WG), August 2006. https://forge.gridforum.org/projects/ogsa-wg.
39. M. Pereira. Resource Namespace Service Specification, 2006. https://forge.gridforum.org/sf/projects/gfs-wg.
40. R. M. Piro, A. Guarise, and A. Werbrouck. An Economy-based Accounting Infrastructure for the DataGrid. In *GRID '03: Proceedings of the Fourth International Workshop on Grid Computing*, pages 202–204, Washington, DC, USA, 2003. IEEE Computer Society.
41. R-GMA: Relational Grid Monitoring Architecture, August 2006. http://www.r-gma.org/.
42. T. Sandholm, P. Gardfjäll, E. Elmroth, L. Johnsson, and O. Mulmo. An OGSA-Based Accounting System for Allocation Enforcement Across HPC Centers. In *ICSOC'04*, pages 279–288, USA, 2004. ACM.
43. T. Sandholm, P. Gardfjäll, E. Elmroth, L. Johnsson, and O.Mulmo. A Service-oriented Approach to Enforce Grid Resource Allocations. *International Journal of Cooperative Information Systems*, 15(3):439–459, 2006.
44. T. Sandholm, K. Lai, J. Andrade, and J. Odeberg. Market-Based Resource Allocation using Price Prediction in a High Performance Computing Grid for Scientific Applications. In *HPDC '06: Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing*, pages 132–143. IEEE, June 2006.
45. SGAS Project Page, June 2006. http://www.sgas.se.
46. J. Shneidman, C. Ng, D. C. Parkes, A. AuYoung, A. C. Snoeren, A. Vahdat, and B. Chun. Why Markets Could (But Don't Currently) Solve Resource Allocation Problems in Systems. In *HotOS-X '05: Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems*, June 2005.
47. O. Smirnova, P. Eerola, T. Ekelöf, M. Ellert, J.R. Hansen, A. Konstantinov, B. Kónya, J.L. Nielsen, F. Ould-Saad, and A. Wäänänen. The NorduGrid Architecture and Middleware for Scientific Applications. *Lecture Notes in Computer Science*, 2657:264–273, 2003.
48. SNAC - Swedish National Allocations Committee, June 2006. http://www.snac.vr.se/.
49. L. Srinivasan and T. Banks. Web Services Resource Lifetime 1.2 (WS-ResourceLifetime), June 2004. http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-03.pdf.
50. L. Srinivasan and J. Treadwell. An overview of service-oriented architecture, web services and grid computing, November 2005. http://devresource.hp.com/drc/technical_papers/grid_soa/SOA-Grid-HP.pdf.
51. Swegrid, June 2006. http://www.swegrid.se.
52. TeraGrid, June 2006. http://www.teragrid.org.
53. Usage Record WG (UR-WG), June 2006. https://forge.gridforum.org/projects/ur-wg/.
54. R. Wolski, J. Brevik, J. S. Plank, and T. Bryan. Grid Resource Allocation and Control Using Computational Economies. In F. Berman, G. Fox, and A. Hey, editors, *Grid Computing: Making The Global Infrastructure a Reality*, chapter 32. John Wiley & Sons, 2003.
55. R. Wolski, J. S. Plank, J. Brevik, and T. Bryan. G-commerce: Market Formulations Controlling Resource Allocation on the Computational Grid. In *International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, April 2001. IEEE.