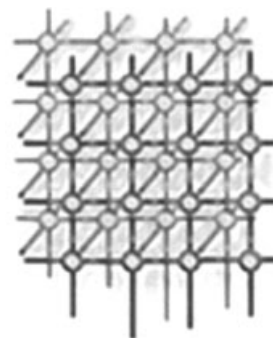# Scalable Grid-wide capacity allocation with the SweGrid Accounting System (SGAS)

Peter Gardfjäll[1], Erik Elmroth[1,*,†], Lennart Johnsson[2], Olle Mulmo[2] and Thomas Sandholm[2]

[1]*Department of Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden*
[2]*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 as a means 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 system design, a flexible allocation model, middleware integration experiences and 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 quota enforcement overhead incurred by SGAS on job submissions is not a limiting factor for the job-handling capacity of the job submission software. Copyright © 2008 John Wiley & Sons, Ltd.

---

*Correspondence to: Erik Elmroth, Department of Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden.
†E-mail: elmroth@cs.umu.se

---

1. INTRODUCTION

As an enabler of large-scale resource sharing, Grid technology promises access to unprecedented computing capacity by integrating pools of computational resources across organizational boundaries, presenting them to users as a single virtual system (a Grid). An important objective of the virtual organization (VO) [1] that shares this computing infrastructure is to make efficient use of the provisioned resource capacity to maintain a high degree of overall system utilization and satisfy individual projects' service needs.

Lacking capacity allocation mechanisms that operate across the Grid, the capacity of most Grid systems to date has 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 overexploitation and degradation of the common resource—a phenomenon often referred to as the 'tragedy of the commons' [2]. 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 these issues with the SweGrid Accounting System (SGAS) [3], a Grid accounting system that tracks usage and enforces Grid-wide usage limits. As such, SGAS serves as a capacity allocation mechanism that coordinates usage across the Grid by logically dividing the aggregate capacity of the VO-provisioned resources between user groups.

For capacity allocation, SGAS employs a credit-based model, expressed in terms of Grid-wide quota allowances that are granted to user groups by an allocation authority. These Grid credit allowances represent an entitled share of the Grid capacity and can be 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 job requests from users who have exceeded their quota. SGAS consists of three main components: a bank service that manages quota accounts, a logging service that publishes Grid usage in a uniform format, and a resource-side job interceptor that reserves quota prior to job execution and charges and logs job usage when the job is completed.

Early prototype work on SGAS was presented in [4,5], and a revised version of [5] was published later in [6]. The main contributions of this article include detailed component descriptions and design rationale, a flexible allocation model based on the notion of time-stamped allocations, experiences from middleware integration with Globus Toolkit 4 (GT4) and the advanced resource connector (ARC) [7], improved scalability via a banking system that is virtualized across several servers, and finally, an extensive performance and scalability evaluation.

Section 2 provides some historical background on SGAS. The requirements and design considerations that underlie the SGAS design are covered in Section 3, and the SGAS architecture, and how it addresses these requirements is presented in Section 4. A more technical description of SGAS is given in Section 5. The performance of SGAS is analyzed in Section 6, and finally, we present related work in Section 7 and some concluding remarks in Section 8.

2. BACKGROUND

The Swedish National Allocation Committee (SNAC) [8] is an allocation authority that controls access to computer clusters at six Swedish HPC centers. SNAC grants computer

time to research groups with substantial computational needs after a peer-reviewed application process[‡].

Traditionally, researchers would apply for computer time on individual machines. Besides the administrative overhead of managing separate usage limits on each machine, this approach also limited users' freedom of choice and caused imbalanced machine load and poor overall utilization of the total capacity.

The emergence of Grid technology provided the technical foundation for interconnecting the HPC centers into a Grid (SweGrid), which allowed all clusters to be treated as a single virtual machine. This not only enabled more flexible resource sharing and load balancing across the machines but also allowed SweGrid to be treated as a single computational resource with common usage limit management. Hence, the previous machine-targeted usage limits were abandoned in favor of Grid-wide allocations that counted towards the Grid as a whole.

SGAS has been developed in response to the need for coordinated enforcement of usage limits across all SweGrid HPC centers. In essence, SGAS extends single-machine quota enforcement (which has been common practice for a long time at HPC centers) to apply to the Grid as a whole. Although extending this concept to Grids may seem simple from a conceptual perspective, (i) the move from local to distributed quota enforcement; (ii) the inherent challenges of Grid environments; and (iii) the conflicts of interest between the three parties that the system serves (user, resource owner, allocation authority) make it a far from trivial exercise. These issues, and how they are addressed, are described in greater detail in the following sections.

Although SGAS was initially targeted towards allocation enforcement within SweGrid, it is designed to be applicable in a wide variety of Grid settings.

3. REQUIREMENTS AND DESIGN CONSIDERATIONS

This section sets out the requirements that underlie the SGAS design. To put these requirements in context we start by describing a conceptual model for the operation context.

3.1. Operation context

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 SweGrid and TeraGrid, where research institutes pool their computational resources. In this operation context, we distinguish three main stakeholders whom our system serves, as illustrated in Figure 1:

- Resource owners, such as the SweGrid HPC centers, who contribute resources to the VO.
- Allocation authority, such as SNAC, which divides the aggregate VO capacity between user groups.

[‡]A similar allocation model is used within the TeraGrid computing facility by the National Resource Allocation Committee (NRAC)—the TeraGrid counterpart of SNAC.
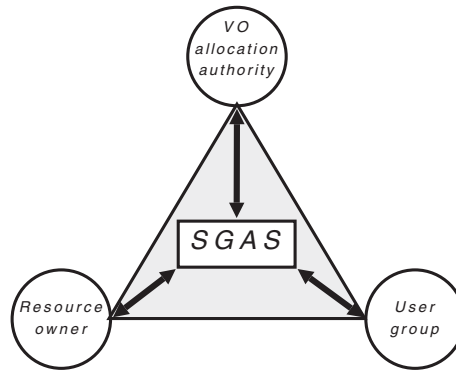
Figure 1. SGAS as the mediator between the parties of the stakeholder triangle.

- Users who consume Grid resources (subject to the restrictions imposed by the allocation authority).

These stakeholders have differing, and to some extent conflicting, interests. The allocation authority wishes to make optimal use of VO capacity and to that end grants capacity allotments to projects on the basis of their computational needs and research contribution. These capacity allocations need to be enforced collectively by the resources. However, with the decentralized management structure of Grids, where owners always retain ultimate control over their resources, such coordination cannot be forced upon resource owners. Strict enforcement of quotas may conflict with resource owners' ambition to achieve high utilization, which typically requires usage limits to be relaxed, thereby improving utilization at the expense of fairness. This may also conflict with users, to whom fairness and quality of service (QoS) guarantees are primary concerns.

In summary, there is a conflict between global and local resource control, and there is a trade-off between resource utilization and allocation enforcement strictness (fairness). SGAS must join these conflicting interests to allow allocation authorities to coordinate usage limit enforcement without sacrificing resource owner autonomy.

3.2. Functional requirements

This section presents the main functional requirements for SGAS. A more detailed description of the system and how it addresses these requirements follow in Sections 4 and 5.

- F1. *Quota/account management*: A key requirement is to manage quota accounts for user groups, which includes managing account membership, controlling allowances, reserving quota and charging jobs for their resource consumption. Furthermore, administrators and authorized account members must be given access to the account credit balance as well as the transaction history.
- F2. *Consistent quota enforcement*: Project allocations must be enforced to prevent users from exceeding their quota allowances. Although different forms of relaxed quota enforcement

models are conceivable, where job payment is deferred, such models would only provide a coarse-grained usage regulation mechanism and would only admit a weak form of overuse protection. The need for strict overuse protection requires a real-time enforcement model where payment is guaranteed for each job prior to execution to prevent users from running jobs without coverage.

F3. *VO-wide usage tracking*: Resource usage must be collected for each job execution in order to determine the cost of the consumed resources. Furthermore, resources should be able to publish local usage of a VO-wide logging facility, which can be queried to present an aggregate view of Grid usage. The logged usage data establishes an audit trail and the possibility to correlate account withdrawals with detailed information about the corresponding job. Since sites may use different internal formats for reporting usage data, usage must to be translated into a uniform usage data format before being published.

F4. *Flexibility*: Given the diversity of Grid resources, the differing needs of VOs, and the conflicts of interest between the system stakeholders (see Section 3.1), SGAS must be extensible and customizable along several dimensions. To this end, SGAS must allow enforcement policy customization to address the need for different degrees of enforcement strictness. For example, it should allow for softer modes of usage limit enforcement to improve local (and Grid-wide) utilization at times of light load. As another example, an allocation authority may wish to introduce a credit limit to allow more flexibility in project spendings (for instance, to account for unpredictable computational needs). Additionally, SGAS must be capable of charging different types of resource usage.

3.3.   Non-functional requirements

The loosely coupled and dynamic nature of Grid environments impose additional requirements on Grid software compared with software written for more tightly coupled cluster computing systems and traditionally (single-organization) distributed computing systems [9]. Such inherent Grid challenges are included in the non-functional requirements, which also need to be addressed by the SGAS design:

N1. *Scale*: Grid systems are typically large-scale systems that integrate widely distributed computational resources that belong to different administrative (and security) domains. Thus, a key requirement for the accounting system is to scale up with large Grid user populations and resource collections, and the request traffic they produce.

N2. *Site autonomy*: Management of Grids is decentralized in nature, with the core principle being that resource owners should always retain ultimate control over their resources. Therefore, all job admission decisions must be made subject to local resource owner policies, allowing resource owners to overrule global allocation policies (decided by the allocation authority) in order to honor the site autonomy of resource contributors.

N3. *Heterogeneity*: The computational resources that are shared by VO participants can be, and often are, heterogeneous in terms of hardware, platform, software stacks and data formats. Since SGAS is intended to be a generic tool it must be carefully designed to take the diversity of Grid environments into account. Therefore, SGAS must not target any specific combination of platform, middleware and scheduling systems but rather aim to be simple

to integrate into existing infrastructure and be non-intrusive on the underlying system. Furthermore, the inherent heterogeneity of Grid environments makes interoperability a critical concern. To this end, SGAS must rely heavily on Grid and Web service standards.

N4. *Cross-organizational security*: Since VO members constitute a wider and less-trusted user community, as compared with tightly coupled systems, security becomes a central issue. Furthermore, the exchange of funding tokens and potentially sensitive data make it even more important that accounting information only be exchanged between trusted entities so as to prevent abusive/malicious usage and privacy violations.

N5. *Transparency*: The system should have minimal impact on client side components, such as job submission software and resource brokers. SGAS should effectively be 'invisible' to these components to minimize interference with the existing infrastructure. Furthermore, the system must only place a marginal additional burden on end-users who, ideally, should remain unaware that the Grid is 'accounting-enabled'.

4. ARCHITECTURE

This section presents an overview of the SGAS architecture, which addresses the requirements presented in the previous section.

SGAS is composed of three main components that operate together across the Grid sites to enforce capacity allocations: a bank service, a logging service (the Logging and Usage Tracking Service—LUTS) and a resource integrator (the Job Account Reservation Manager—JARM).

The bank service is a key component that manages project quotas (Req. F1) allocated by the allocation authority and maintains a consistent view of the resources consumed by each project in order to coordinate quota enforcement across the Grid sites. The 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 LUTS tracks usage across the Grid (Req. F3), by allowing Grid sites to publish detailed information about completed jobs in a uniform XML-based format, as prescribed by the Usage Record standard [10]. These usage records can be queried to give an aggregate view of Grid usage.

JARM integrates local resources into the Grid-wide accounting context of SGAS. It enforces quotas and protects against overuse at job submission time (Req. F2) by intercepting each job request and performing admission control based on the credit balance of the job submitter. JARM enforces quotas strictly by guaranteeing funding availability for each job prior to execution via a pre-acceptance quota reservation against the job submitter's bank account. Failure to make a reservation would indicate overuse and the job can be denied access. Hence, resources enforce allocations in a collective effort coordinated by the bank. However, all job admission decisions are subject to local resource owner policies, which allows bank decisions to be overruled by JARM (Req. N2). 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 (Req. F4). The rationale behind this is that the resource owner always retains ultimate control over the degree of usage limit enforcement applied to local resources. Under a strict enforcement policy, all quota-exceeding jobs are disallowed. Under a softer enforcement

mode, resource owners may allow quota-exceeding jobs to improve local utilization at times of light load. As discussed in Section 3.1, the degree of enforcement strictness represents a trade-off between the degree of overuse protection (fairness) and utilization.

Furthermore, by reserving and charging the usage on behalf of users (using job submitters' delegated credentials) JARM also acts as an enabler of client-side transparency (Req. N5), since the client is never directly involved in accounting system interactions.

The modular structure of JARM, which provides plugin points for customizations and middleware-specific adapter code, simplifies JARM integration with different combinations of middleware platforms, scheduling systems and usage data formats, thereby addressing Req. N3. To date, JARM has been integrated with two middlewares: Globus Toolkit 4 (GT4) and the ARC, both of which may run on top of different scheduling systems with different usage data formats.

The allocation credits are unitless and can hence be used to charge for arbitrary resource usage (only CPU time is currently charged in SweGrid). 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, which would allow resources to charge for different combinations of resource usage. JARM provides plugin points for incorporating different pricing models and calculating usage cost. Accounting for multiple resource types (Req. F4) can either be accomplished by maintaining separate quota accounts for each resource type or by charging a single account for combinations of different resource types (via a type-aware pricing scheme). The preferable solution depends on the accounting needs of the organization.

The decentralized enforcement model of SGAS (with resources cooperating to enforce quotas) together with the ability to distribute the bank component across several branch servers to balance load (see Section 5.5) addresses the issue of scalability (Req. N1).

There are several reasons why the bank and the logging service are kept separated. This approach follows the 'separation of concerns' design principle. This design principle improves system modularity, simplifies reuse and also helps in distributing load in the system (hence addressing the scalability requirement). Furthermore, it enables SGAS to operate in logging-only mode (only employing the logging service), which is useful in environments where usage only needs to be logged and not charged.

Since both the Bank and the LUTS operate as remote services, they are completely independent of the middleware being deployed in the targeted Grid environment, whereas a small piece of adapter code is needed to integrate JARM with new Grid software (Req. N3).

Finally, the security/trust issue (Req. N4) is addressed by an interoperable end-to-end security infrastructure that makes heavy use of security standards for authentication and authorization (see Section 5.4).

4.1. SGAS operation

This section presents an overview of the system operation, including how allocation policies are set up and how components interact to enforce usage limits. These interactions are illustrated in Figure 2. A more technical description of these components is given in Section 5.

Before allocations can be enforced, the allocation authority needs to decide upon an allocation policy that divides capacity between projects (1). These capacity allotments, which are granted
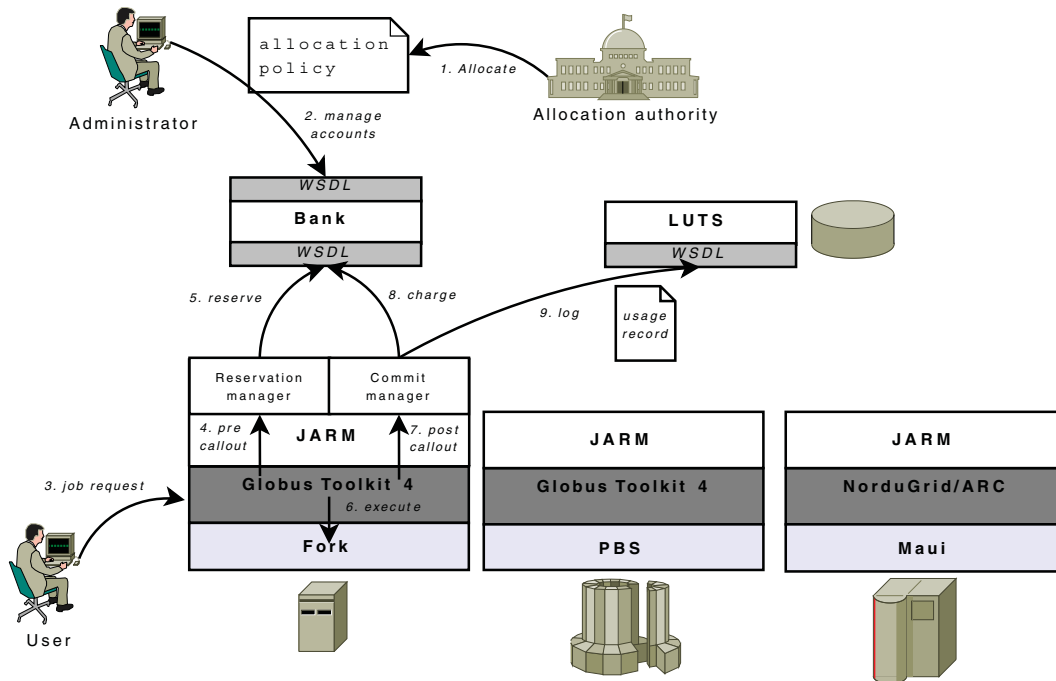
Figure 2. SGAS system overview.

to users as credit allowances that, for instance, may represent computer time entitlements, are deposited to the bank accounts by a bank administrator (2).

When a job request arrives at a resource (3), it is intercepted by JARM via a middleware call-out (4). Before allowing the job to be executed, JARM makes a quota reservation on the job submitter's bank account (5) to guarantee job funding prior to execution, thereby enforcing project allocation and preventing overuse. This quota reservation is made on behalf of the user with the job submitter's delegated credentials and is therefore transparent to client-side components. If the reservation is successful the job is allowed to be executed (6). Otherwise the job may be refused, depending on the resource's degree of enforcement strictness. After the job is executed, another call-out is made to JARM (7), which collects usage data for the job, calculates the cost of the job, charges the account with the previously acquired reservation (8) and logs a usage record in the logging service (9). JARM needs to transform usage from the local format used by the resource to the uniform usage record format before logging to the LUTS.

4.2. 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 prevented from overdrawing their accounts. However, the volatile nature of CPU time causes a subtle problem with the credit-based allocation model, which may lead to situations where users are unable to spend their entire allocations.

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 modeled capacity (represented by the credits in circulation). That is, inactive projects save credits that, eventually, will not have any correlation to actual CPU time, since the CPU cycles that they represent have already been lost. This may cause contention problems at allocation period borders, when all projects attempt to spend any 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 some projects are unable to spend their quotas.

SGAS partially solves, or at least reduces, this 'inflation' problem through the use of time-stamped allocations. Each account has a 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 test runs 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 credits–capacity correlation. Although long validity periods allow more flexibility in quota spending (over time) and hence may improve overall utilization, they increase the risk of quota accumulation and period-end contention. Short allocation periods, on the other hand, facilitate closer credits–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 strategy when issuing resource grants. The under-provisioning approach (less issued credits than actual capacity) may lead to poor utilization since not all users may consume their full share, whereas contract/quota fulfillment and fairness becomes easier to deliver. In the overbooked case (more issued credits than actual capacity), 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 credits–capacity balancing price scheme, it would be straightforward to plug into JARM (which supports custom pricing schemes, see Section 5.3) once it is available.

Competitive, market-based pricing schemes have been proposed in the literature [11–16] as a means of balancing load between resources (by attracting users to lightly loaded resources with low prices and vice versa) and achieving service differentiation (users pay more to receive better QoS). One such approach was demonstrated in [17], with minimal impact on the middleware but with a completely different resource provisioning model (based on virtualization). See Section 7 for a continued discussion on market-based resource allocation.

5. DESIGN AND IMPLEMENTATION

This section covers the SGAS components in greater technical detail.

Since interoperability is a critical concern in Grid environments (as an enabler of seamless operation across the heterogeneous resource base), we have adopted the standard service-oriented system principles proposed by the Global Grid Forum (GGF) [18] through the Open Grid Service Architecture (OGSA) [19]. 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) [20] 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.

The SGAS software package, which is available under an open-source license, can be downloaded from the SGAS web site [3]. SGAS has been developed using GT4 (Java WS Core) [21], a Web service development framework that adds WSRF primitives to the Axis Web service engine [22]. These primitives can be combined to build OGSA-compliant Web services. The implementation is entirely Java based, which results in a portable code that runs on any platform with Java support. SGAS is currently included as a technology preview in GT4.

The sequence diagram in Figure 3, which shows the typical interactions involved in a job request, may be useful as a reference while reading the design section.

5.1. Bank

The bank is composed of three WSRF-compliant Web services (bank, account, hold), whose relationships are shown in Figure 4. 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 [23]. The creation of an account produces an account WS-resource repre-

---

[§]SweGrid uses a static pricing scheme with a fixed exchange rate of one credit per wall clock CPU second.
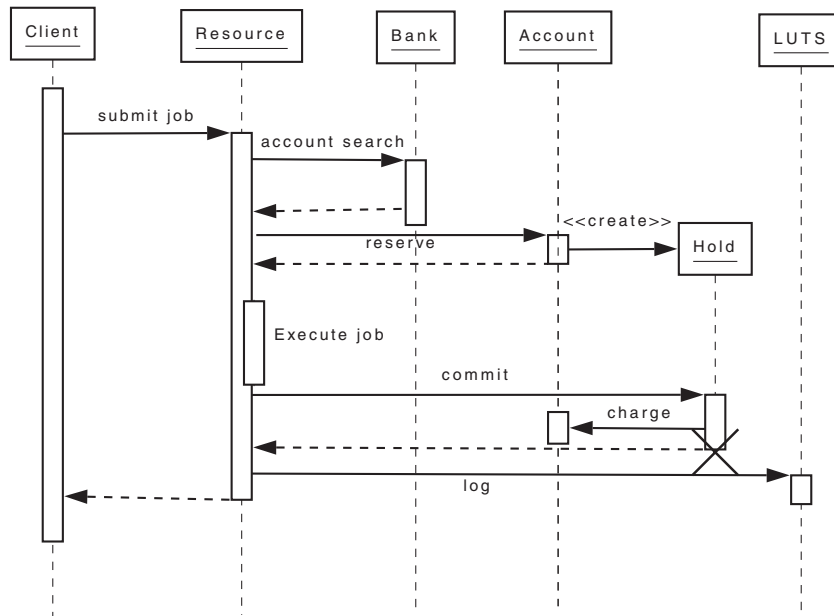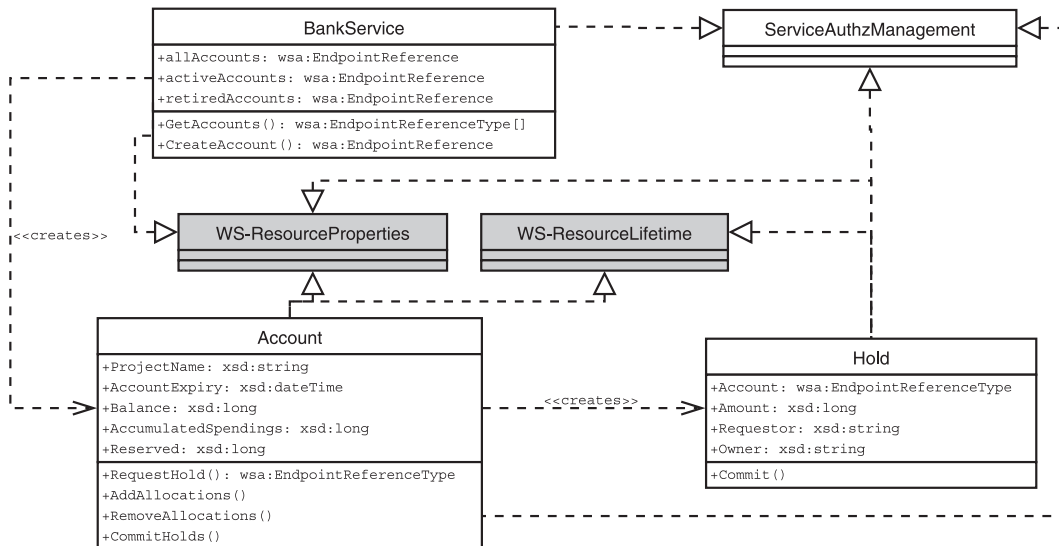
Figure 3. Accounting interactions.

Figure 4. Bank services.

senting the project allocation. 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) [24] 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 account service interface contains operations for managing the project allocation (adding and removing time-stamped allocations) 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 5.4) upon which all SGAS services rely (the ServiceAuthzManagement interface). Account members reserve a portion of the account allocation for each job request. 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 reservation 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 and improves overall scalability 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 violate the credit limit. However, the bank decision can be overruled by JARM, which, depending on local site policies, may still allow the job to be executed, for example, to improve local utilization.

All services publish their state via the get/query operations provided by the WS-Resource-Properties 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 [26], 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 5.5, the bank is not necessarily 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'.

5.2. Logging and usage tracking service

The LUTS provides a Web service interface for publishing usage data in the uniform format prescribed by GGF-UR [10] and for query-based retrieval of usage data using the XPath query language [27].
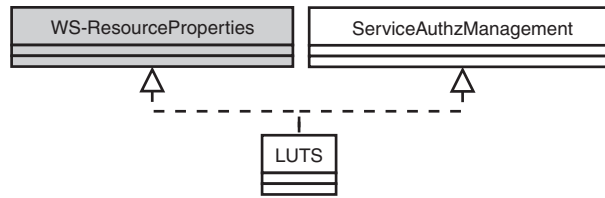
Figure 5. Logging and usage tracking service (LUTS) interface.

The service interface, shown in Figure 5, 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 ServiceAuthz-Management rights administration interface, allowing differentiated publish (more restricted) and query (more generous) access rights.

SGAS uses a native XML database (eXist [28]) for persistent storage and recoverability of usage records. We provide a query dialect, implemented as a custom query expression evaluator [29], 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 XML documents are stored 'as is' in the database) facilitate extensibility, which allows sites to publish custom usage record elements without modifying the LUTS.

5.3. Job account reservation manager

JARM is a piece of integration software that is plugged into existing middleware stacks to enforce quotas on the resources by reserving and charging allocations on behalf of job submitters.

An overview of JARM's job submission handling is illustrated 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 the job's resource 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's (deny) decision. Permitting jobs to run in spite of 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. Excessively 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.

5.3.1. Integration overview

Figure 6 illustrates JARM's components (in terms of Java interfaces). The figure also shows JARM's 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 usage limits. 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.
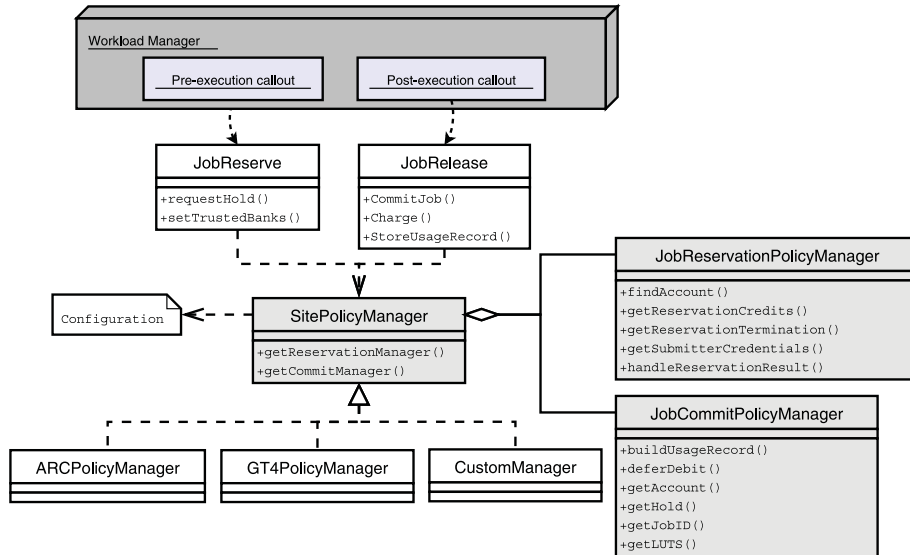
Figure 6. JARM components.

5.3.2. Integration approaches

Different approaches are conceivable for the pre-/post-execution call-outs. The NorduGrid/ARC [7] 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 7, 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 [30] 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 need to be collected from
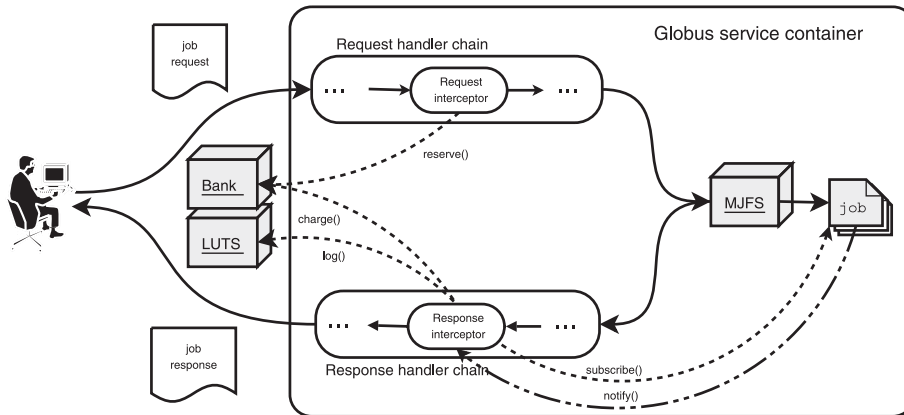
Figure 7. WS-GRAM integration.

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 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.

5.4. Security

Since SGAS security is extensively covered in [6] and remains essentially unchanged; this section only provides cursory coverage of the SGAS security solution.

SGAS offers flexible and interoperable end-to-end security 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 [31], WS-SecureConversation [32]) and transport-level security (via TLS [33]), as provided by GT4.

Another part of the SGAS security solution is a service-orthogonal authorization framework that allows run-time administration of XML-based authorization policies through a Web service interface (ServiceAuthzManagement), as shown in Figure 8. This authorization interface provides plug-in support for different back-end authorization engines. The current back-end is based on the eXtensible Access Control Markup Language (XACML) [34] 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
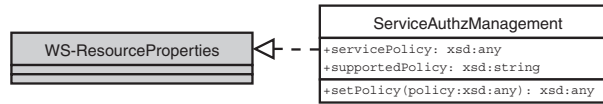
Figure 8. ServiceAuthzManagement (SAM) service interface.

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).

5.5.   Service distribution and the virtual bank

As discussed in Section 3.3, scalability is a key challenge in Grid environments. Being the central coordinator of quota enforcement, the bank could quickly become a performance bottleneck in large-scale Grid settings, as the user and resource populations grow with an accompanying increase in job submission and account request rate. The bank's load capacity can be improved by distributing the bank accounts across several bank branch servers. However, a distributed bank needs to be carefully designed. Simply deploying a set of bank servers that manage separate subsets of accounts has several drawbacks.

- Bank administrators would need to manage each branch separately, including keeping track of the server addresses as well as which accounts are hosted on what servers.
- Users who are members of several accounts would need to keep track of the physical addresses of their account branch servers, in order to target a specific account for job payment (in the job request).
- Resource owners would need to reconfigure their resource to explicitly maintain lists of trusted bank servers (and their host certificates), as well as user-to-branch mappings for all users, directing each user to its (set of) branch(es). This task quickly becomes tedious, especially considering a large and constantly changing user base and that such settings would need to be reconfigured whenever a branch server is relocated.

To prevent such an administration nightmare, a more dynamic and elegant solution is required that allows the bank to be partitioned across several servers, while preserving the illusion of a single virtual bank service. Such a solution would allow dynamic (and transparent) provisioning of more branch servers to adapt to VO growth.

The key enabler of the virtual bank is an abstract-naming scheme, which introduces an extra level of naming indirection where accounts are referred to via logical (location-independent) names, which are resolved by a naming service to the physical address prior to invocation (akin to URL to IP address translation in the DNS). This naming scheme adds value in several respects:

- It produces scaling transparency, which allows additional branch servers to be introduced, while still presenting a single logical service to clients.
- It produces location transparency, which allows users to disregard the physical locations of branches/accounts. This facilitates server relocations, which, for example, may be necessitated by machine park maintenance or upgrades.

- It simplifies the lives of end-users who can refer to accounts via abstract names like sgas://atlas-account instead of physical addresses in their job requests.
- It allows automatic trust establishment with new branches without requiring resource owner intervention.

The 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 the name service, are not SGAS-specific and could thus be useful outside the SGAS context as well.

5.5.1. Name 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 the other for name resolution.

The name registration solution is inspired by the resource namespace service (RNS) specification [35]. We chose not to 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 [36] 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 [24]). The WSRF-based service interfaces that constitute the name service are shown in Figure 9.

Name registration: The registration part of the name service manages hierarchically arranged collections of many-to-many name-to-address mappings. It consists of 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 modeled 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 [26]), 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.

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
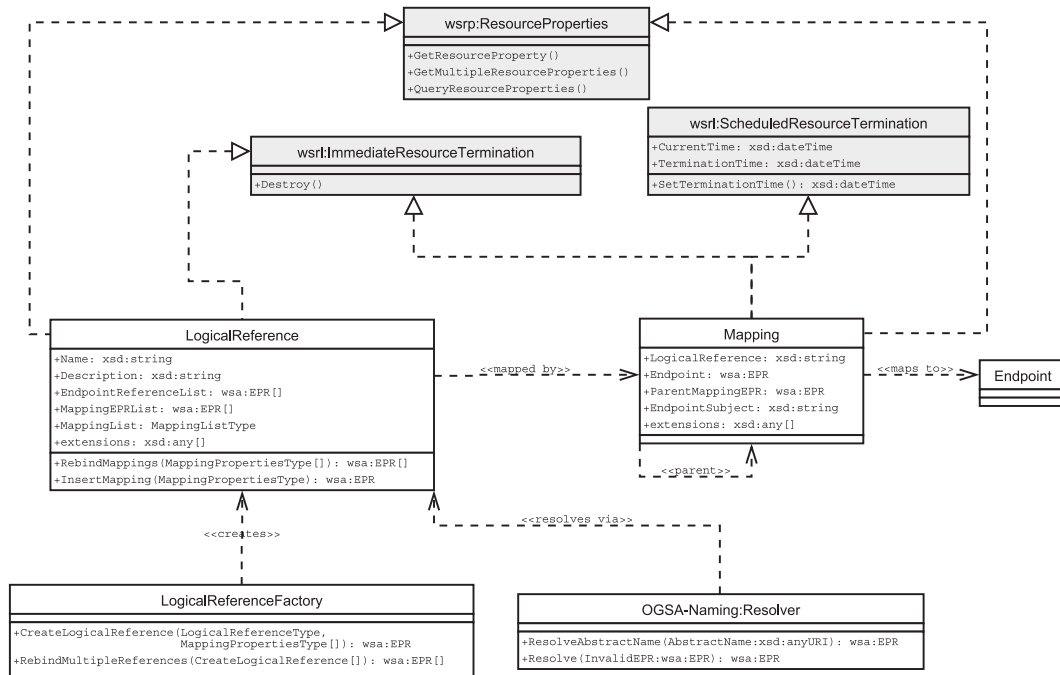
Figure 9. The name service interfaces.

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 the other 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, thus 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.

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.

5.5.2. *Virtualizing the bank*

With the name service infrastructure in place, implementing the virtual bank becomes straightforward.

In the distributed virtual bank configuration, each bank becomes a bank branch responsible for a separate subset of accounts. This is illustrated in Figure 10, 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 EPR 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
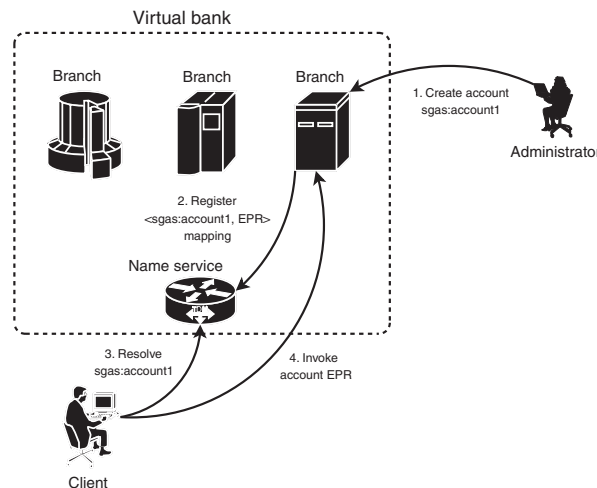

Figure 10. A virtual bank distributed across several hosts.

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 (authorized) target. Trust with new branches will then be automatically established at resolution time, by dynamically adding the identity of the resolved endpoint to the set of trusted identities for the invocation.

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, to support the full range of possible queries, such a solution requires distributed query processing as well.

6. 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).

6.1. Testbed

During the tests, two separate sets of computers were used, both connected through a $100\,\text{Mbit s}^{-1}$ 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 16 of them were equipped with Intel Pentium 2.8 GHz processors and 1 GB memory, while the other 16 had AMD Athlon 64 2.0 GHz dual core processors with 2 GB memory.

The DiPerf performance testing framework [37] was used to drive and coordinate the test clients, collect time measurements and compile the results into a single (global) time-line. 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.

6.2. 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.

6.2.1. 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 s, and each client issues requests for 40 min. The target account is chosen at random from the accounts hosted by the bank.

The results are shown in Figure 11, 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 min. 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 min

---

[‖]This functionality is provided as part of the GT4.1.0 development release.
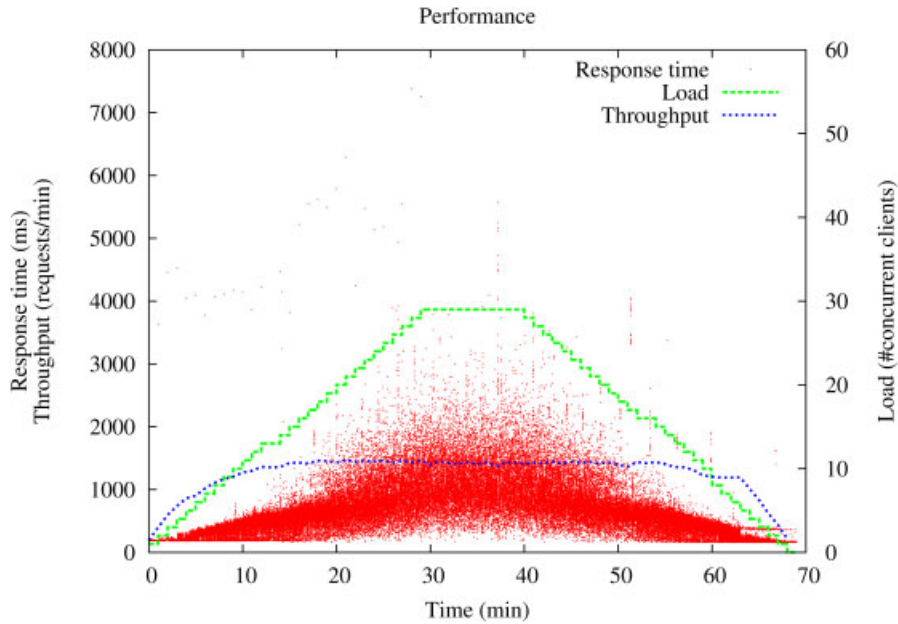
Figure 11. Reservation requests against the bank.

a throughput limit is reached (about 1400 requests min^{-1}). 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 min 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 min^{-1}.

During the test a total of 86 584 reservation requests were issued. Given the test duration of 70 min, 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 min^{-1}, or about 23 requests s^{-1}. 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.

6.2.2. 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 s). 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 12, with axes as described in Section 6.2.1.

We can see that the maximum throughput in the virtual bank case is reached after about 20 min (40 clients). The throughput limit in this case is roughly 2700 requests min^{-1}, which is close to a doubling of throughput from the single branch case (1400 requests min^{-1}). 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 of each other, the two branches are capable of handling twice the load of a single bank, with twice as high throughput.
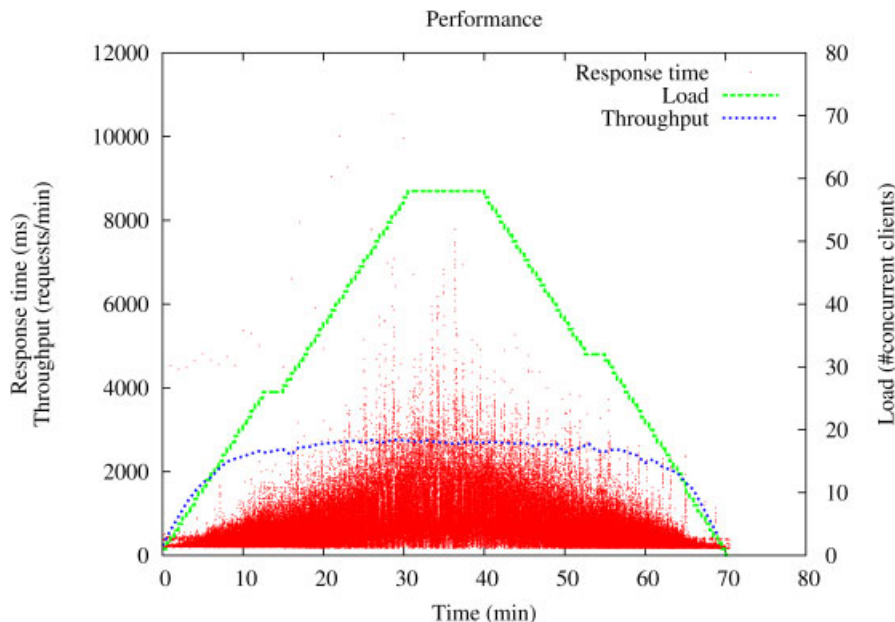

Figure 12. Reservation requests against a virtual bank with two branches.

Similar improvements should be observed for each added branch, given that load is sufficiently well balanced between the branches.

6.2.3. 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. All the tests follow the same general pattern: eight simultaneously started tester clients submit jobs to a GRAM container for a 20 min 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, whereas two 'accounting-enabled' containers (with JARM intercepting jobs) were tested: one with the regular SGAS version and another 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 be completed 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 13 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

Table I. The submission options.

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

Table II. GRAM protocol steps.

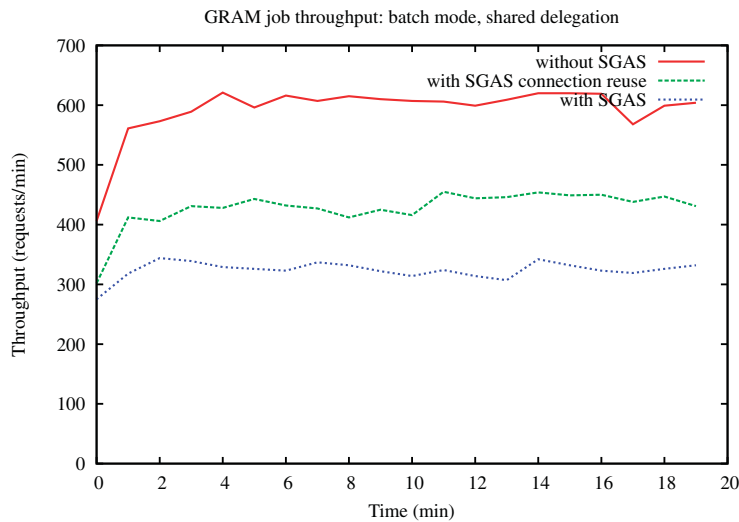| Protocol step	B/S	B/P	I/S	I/P
Delegate proxy		√		√
Submit job	√	√	√	√
Execute job			√	√
State notifications			√	√
Job cleanup			√	√
Proxy cleanup				√

Figure 13. Streamlined (batch mode, shared delegation) job submissions.

this submission mode uses. Each client delegates a single proxy credential that is shared by all jobs (shared mode) and no job state change subscriptions are established (batch mode). As can be seen from the other GRAM figures (14, 15, 16), 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 $\min^{-1}$ on average, whereas the regular SGAS integration reduces the throughput to half (320 requests $\min^{-1}$). The connection reuse SGAS version offers a significant improvement in this case, handling on average 430 requests $\min^{-1}$. The main source of SGAS overhead is the account reservation invocation performed prior to accepting each job submission.

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 10 arriving jobs per second is unlikely to be observed on a real production environment Grid resource; 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 $\min^{-1}$.

Figure 14 illustrates the throughput numbers when per-job delegation has been added to the streamlined job submissions of Figure 13. 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 $\min^{-1}$ without SGAS, 65 jobs $\min^{-1}$ with SGAS and 70 jobs $\min^{-1}$ 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 15. Here, clients delegate separate proxy credentials for each job (per-job delegation) and also await 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 $\min^{-1}$ in the pure GRAM case, 50 jobs $\min^{-1}$ for SGAS-enabled GRAM and 53 jobs $\min^{-1}$ for SGAS with connection reuse.

The steadily decreasing throughput curves that can be observed in the per-job delegation test cases (Figures 14 and 15) 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 16. 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 $\min^{-1}$, whereas the SGAS-enabled GRAM manages 84 jobs $\min^{-1}$. In this case, the connection reuse SGAS version case is very close to the
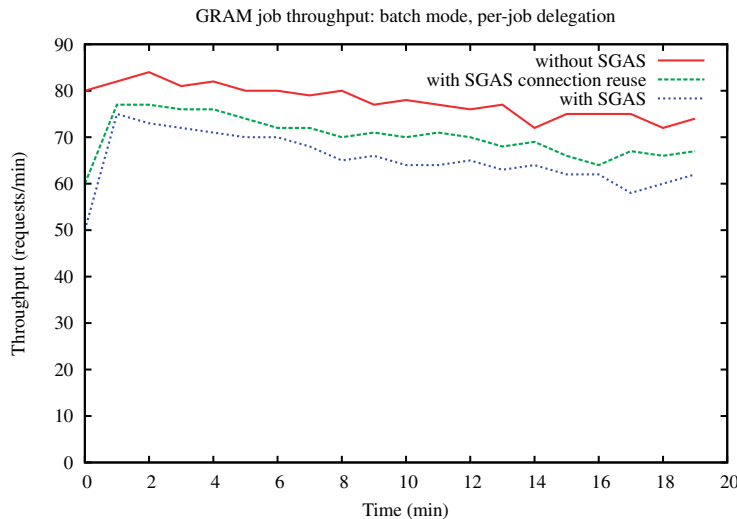

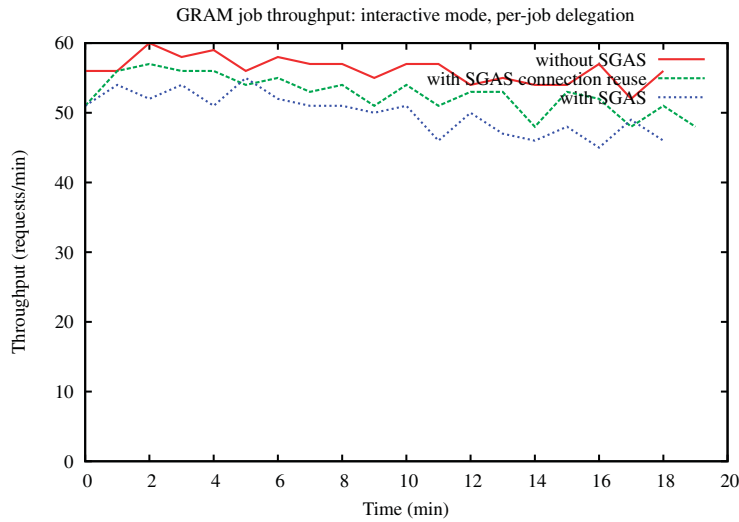Figure 14. Batch mode, per-job delegation job submissions.

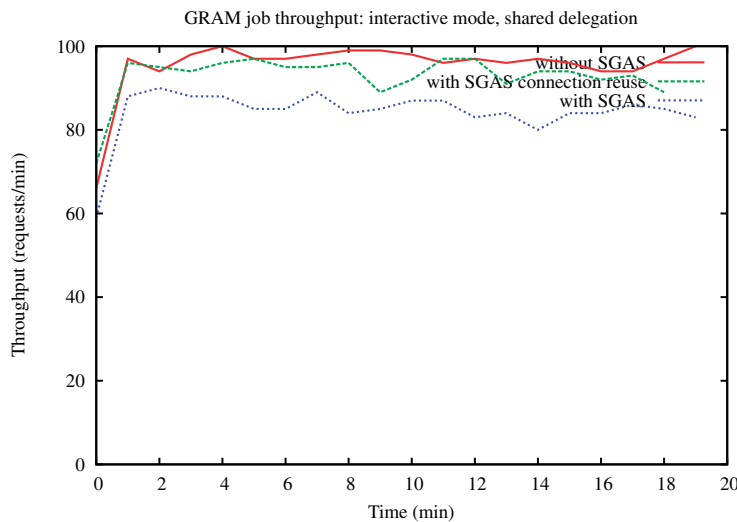Figure 15. All-inclusive (interactive mode, per-job delegation) job submissions.

Figure 16. Interactive mode, shared delegation job submissions.

unaltered GRAM with 93 jobs min^{-1}. Judging by the sheer number of message exchanges involved in the different submission modes, one would assume that the B/P mode (Figure 14) would allow higher throughput than the I/S mode (Figure 16). 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).

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

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 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 incurs around 10–15% overhead on GRAM job submissions. 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). It should be noted, however, that all these tests put the system under extreme stress, especially considering that all load is put on a single Grid resource with the lightest load being one new job per second. Even this job arrival rate is not likely to be seen on Grid resources within a foreseeable future. In particular, the batch/shared test case represents an extreme scenario that pushes the limits of the system with a job arrival rate of $10\,\text{jobs}\,\text{s}^{-1}$, 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.

7. RELATED WORK

Compared with the widely distributed and heterogeneous nature of Grid environments, resource allocation and usage tracking in HPC/cluster computing environments are 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, whereas other systems allow market forces allocate capacity via supply and demand driven interactions.

Gold [38] 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 [39], which was aimed at single institution accounting.

The distributed Grid accounting system (DGAS) [40] is an accounting system developed within the context of the EGEE project [41], 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 [42], thus limiting its scope of applicability.

GridBank [43] 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 [44] 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 a (non-standard) relational data format as prescribed by the R-GMA information system [45]. A similar approach is taken by MOGAS [46], which collects usage data and stores it in a relational database for publishing of different kinds of Grid usage statistics on the web.

A share-based approach to allocate Grid capacity has been proposed by Dumitrescu and Foster [47] and Elmroth and Gardfjäll [48]. 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 [48], 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 [47].

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 [11–16,49].

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 [11,13–15]; (2) it promotes user-centric scheduling with per-task QoS differentiation [13–15]; (3) market prices regulate resource supply and demand towards a state of market equilibrium where supply and demand are at balance [11–13,49], and

(4) markets operate in a decentralized[||] and efficient manner, without need for centralized control [13,49].

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 [50]. 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 [51] 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 [52]. 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.

Finally, related work also includes two recent investigations and comparisons of existing Grid accounting systems. In [53], SGAS was deemed superior to the six other systems investigated (APEL, DGAS, GridBank, GRASP [54], GSAX [55], and Nimrod/G [56]), fulfilling nearly twice as many of the evaluation criteria compared with any other system. In another accounting system survey [57], comparing the same seven accounting systems, SGAS was also found to fulfill the largest number of requirements.

8. 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

---

[||]Agents acting in self-interest achieve global 'welfare'.

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 modeled 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. The SGAS software package, which is currently included as a technology preview in GT4, can be downloaded from the SGAS web site [3].

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 min^{-1}, 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.

REFERENCES

1. Foster I, Kesselman C, Tuecke S. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications* 2001; **15**(3):200–222.
2. Hardin G. The tragedy of the commons. *Science* 1968; **162**(3859):1243–1248.

3. SGAS Project Page. June 2006. http://www.sgas.se [4 January 2008].
4. Elmroth E, Gardfjäll P, Mulmo O, Sandholm T. An OGSA-based bank service for grid accounting systems. *Applied Parallel Computing* (*Lecture Notes in Computer Science*). Springer: Berlin, 2006; 1051–1060.
5. Sandholm T, Gardfjäll P, Elmroth E, Johnsson L, Mulmo O. An OGSA-based accounting system for allocation enforcement across HPC centers. *ICSOC'04*, U.S.A., 2004. ACM: New York, 2004; 279–288.
6. Sandholm T, Gardfjäll P, Elmroth E, Johnsson L, Mulmo O. A service-oriented approach to enforce grid resource allocations. *International Journal of Cooperative Information Systems* 2006; **15**(3):439–459.
7. Smirnova O, Eerola P, Ekelöf T, Ellert M, Hansen JR, Konstantinov A, Kónya B, Nielsen JL, Ould-Saad F, Wäänänen A. *The NorduGrid Architecture and Middleware for Scientific Applications* (*Lecture Notes in Computer Science*, vol. 2657), 2003; 264–273.
8. SNAC—Swedish National Allocations Committee. June 2006. http://www.snac.vr.se/ [4 January 2008].
9. Srinivasan L, Treadwell J. An overview of service-oriented architecture, web services and grid computing. November http://h71028.www7.hp.com/ERC/downloads/SOA-Grid-HP-WhitePaper.pdf [4 January 2008].
10. Usage Record WG (UR-WG). June 2006. https://forge.gridforum.org/projects/ur-wg/ [4 January 2008].
11. Piro RM, Guarise A, Werbrouck A. An economy-based accounting infrastructure for the DataGrid. *GRID '03*: *Proceedings of the Fourth International Workshop on Grid Computing*, Washington, DC, U.S.A. IEEE Computer Society: Silver Spring, MD, 2003; 202–204.
12. Wolski R, Plank JS, Brevik J, Bryan T. G-commerce: Market formulations controlling resource allocation on the computational grid. *International Parallel and Distributed Processing Symposium* (*IPDPS*), San Francisco, April 2001. IEEE Press: New York, 2001.
13. Buyya R, Abramson D, Giddy J. An economy driven resource management architecture for global computational power grids. *The International Conference on Parallel and Distributed Processing Techniques and Applications* (*PDPTA*), Las Vegas, June 2000.
14. Chun BN, Culler DE. Market-based proportional resource sharing for clusters. *Technical Report CSD-1092*, Computer Science Division, University of California at Berkeley, January 2000; 1–19.
15. Lai K. Markets are dead, long live markets. *Technical Report*, HP Labs, Palo Alto, CA, U.S.A., February 2005; 1–5.
16. Lai K, Rasmusson L, Adar E, Sorkin S, Zhang L, Huberman BA. Tycoon: An implementation of a distributed market-based resource allocation system. *Technical Report arXiv:cs.DC/0412038*, HP Labs, Palo Alto, CA, U.S.A., December 2004; 1–26.
17. Sandholm T, Lai K, Andrade J, Odeberg J. Market-based resource allocation using price prediction in a high performance computing grid for scientific applications. *HPDC '06*: *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing*. IEEE Press: New York, 2006; 132–143.
18. Global Grid Forum. August 2006. http://www.ggf.org [4 January 2008].
19. Open Grid Services Architecture WG (OGSA-WG). August 2006. https://forge.gridforum.org/projects/ogsa-wg [4 January 2008].
20. Globus: WSRF—The WS-Resource Framework. August 2006. http://www.globus.org/wsrf/ [4 January 2008].
21. Foster I. Globus Toolkit Version 4: Software for service-oriented systems. *IFIP International Conference on Network and Parallel Computing* (*Lecture Notes in Computer Science*, vol. 3779). Springer: Berlin, 2005; 2–13.
22. Apache Web Services Project—Axis. August 2006. http://ws.apache.org/axis [4 January 2008].
23. Foster I, Frey J, Graham S, Tuecke S, Czajkowski K, Ferguson D, Leymann F, Nally M, Sedukhin I, Snelling D, Storey T, Vambenepe W, Weerawarana S. Modeling Stateful Resources with Web Services. 2004. http://www-128.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf [4 January 2008].
24. Gudgin M, Hadley M. Web Services Addressing—Core. December 2004. http://www.w3.org/TR/2004/WD-ws-addr-core-20041208/ [4 January 2008].
25. Graham S, Treadwell J. 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 [4 January 2008].
26. Srinivasan L, Banks T. 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 [4 January 2008].
27. Clark J, DeRose S. XML Path Language (XPath) Version 1.0. November 1999. http://www.w3.org/TR/xpath [4 January 2008].
28. eXist—Open Source Native XML Database. August 2006. http://exist.sourceforge.net/ [4 January 2008].
29. Gawor J, Meder S. GT4 WS Java Core Design. November 2004. http://www.globus.org/toolkit/docs/4.0/common/javawscore/developer/JavaWSCoreDesign.pdf [4 January 2008].
30. Graham S, Murray B. 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 [4 January 2008].
31. Nadalin A, Kaler C, Hallam-Baker P, Monzillo R. 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 [4 January 2008].

32. Gudgin M, Nadalin A. Web Services Secure Conversation Language (WS-SecureConversation). February 2005. http://specs.xmlsoap.org/ws/2005/02/sc/WS-SecureConversation.pdf [4 January 2008].

33. Dierks T, Allen C. The TLS Protocol Version 1.0. January 1999. http://tools.ietf.org/html/rfc2246 [4 January 2008].

34. Godik S, Moses T. 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 [4 January 2008].

35. Pereira M. Resource Namespace Service Specification. 2006. https://forge.gridforum.org/sf/projects/gfs-wg [4 January 2008].

36. Grimshaw A, Snelling D. WS-Naming Specification. September 2006. https://forge.gridforum.org/sf/projects/ogsa-naming-wg [4 January 2008].

37. Dumitrescu C, Raicu I, Ripeanu M, Foster I. DiPerF: An Automated DIstributed PERformance Testing Framework. *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing* (*GRID'04*), Washington, DC, U.S.A., 2004. IEEE Computer Society: Silver Spring, MD, 2005; 289–296.

38. Jackson S. The Gold Accounting and Allocation Manager. 2006. http://sss.scl.ameslab.gov/gold.shtml.

39. Jackson S. QBank—Allocation Manager. 2006. http://sss.scl.ameslab.gov/qbank.shtml.

40. Distributed Grid Accounting System (DGAS). August 2006. http://www.to.infn.it/grid/accounting/ [4 January 2008].

41. Enabling Grids for E-sciencE. August 2006. http://www.eu-egee.org/ [4 January 2008].

42. LCG Middleware. August 2006. http://lcg.web.cern.ch/LCG/activities/middleware.html [4 January 2008].

43. Barmouta A, Buyya R. A grid accounting services architecture (GASA) for distributed systems sharing and integration. *IPDPS'03*, France, 2003.

44. Byrom R, Cordenonsi R, Cornwall L, Craig M, Djaoui A, Duncan A, Fisher S, Gordon J, Hicks S, Kant D, Leakec J, Middleton R, Thorpe M, Walk J, Wilson A. APEL: An implementation of grid accounting using R-GMA. *U.K. e-Science All Hands Conference*, Nottingham, U.K., September 2005.

45. R-GMA: Relational Grid Monitoring Architecture. August 2006. http://www.r-gma.org/ [4 January 2008].

46. Lee B-S, Tang M, Zhang J, Soon OY, Zheng C, Arzberger P, Abramson D. Analysis of Jobs in a multi-organizational grid test-bed. *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and Grid Workshops*. IEEE Press: New York, 2006.

47. Dumitrescu C, Foster I. Usage policy-based CPU sharing in virtual organizations. *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing* (*GRID'04*), Washington, DC, U.S.A., 2004. IEEE Computer Society: Silver Spring, MD, 2004; 53–60.

48. Elmroth E, Gardfjäll P. Design and evaluation of a decentralized system for grid-wide fairshare scheduling. *e-Science 2005*: *First International Conference on e-Science and Grid Computing*, Washington, DC, U.S.A., 2005. IEEE Computer Society: Silver Spring, MD, 2005; 221–229.

49. Ardaiz O, Artigas P, Eymann T, Freitag F, Navarro L, Reinicke M. The catallaxy approach for decentralized economic-based allocation in grid resource and service markets. *Applied Intelligence* 2006; **25**(2):131–145.

50. Wolski R, Brevik J, Plank JS, Bryan T. Grid resource allocation and control using computational economies. *Grid Computing*: *Making The Global Infrastructure a Reality*, Chapter 32, Berman F, Fox G, Hey A (eds.). Wiley: New York, 2003.

51. Nakai J, Van Der Wijngaart RF. Applicability of markets to global scheduling in grids. *Technical Report NAS-03-004*, NASA Advanced Supercomputing (NAS) Division, February 2003; 1–37.

52. Shneidman J, Ng C, Parkes DC, AuYoung A, Snoeren AC, Vahdat A, Chun B. Why markets could (but don't currently) solve resource allocation problems in systems. *HotOS-X '05*: *Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems*, Santa Fe, NM, June 2005.

53. Rückemann C-P. Müller W, von Voigt G. Comparison of grid accounting concepts for D-grid. *Proceedings of Cracow Grid Workshop 06*, Cracow, October 2006.

54. Dimitrakos T, Randal DM, Yuan F, Gaeta M, Laria G, Ritrovato P, Serhan B, Wesner S, Wulf K. An emerging architecture enabling grid-based application service provision. *Proceedings of EDOC.03*, *7th IEEE International Enterprise Distributed Object Computing*. IEEE Press: New York, 2003.

55. Beardsmore A, Hartley K, Hawkins S, Laws S, Magowan J, Twigg A. GSAX Grid Service Accounting Extensions. September 2002. http://www.doc.ic.ac.uk/~sjn5/GGF/ggf-rus-gsax-01.pdf [4 January 2008].

56. Nimrod/G web site. August 2007. http://www.csse.monash.edu.au/~davida/nimrod/nimrodg.htm [4 January 2008].

57. Göhner M, Waldburger M, Gubler F, Rodosek GD, Stiller B. An accounting model for dynamic virtual organizations. *Technical Report No. 2006.11*, University of Zürich, Department of Informatics, November 2006; 1–13.